RICE UNIVERSITY

Scheduling Macro-DataFlow Programs on Task-Parallel Runtime Systems

by

Sağnak Taşırlar

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Vivek Sarkar
Professor of Computer Science
E.D. Butcher Chair in Engineering

Keith D. Cooper
L. John and Ann H. Doerr Professor of
Computational Engineering

Lin Zhong
Assistant Professor of Electrical and
Computer Engineering and Computer
Science

Zoran Budimlić
Research Scientist

Houston, Texas

April, 2011

ABSTRACT

Scheduling Macro-DataFlow Programs on Task-Parallel Runtime Systems

by

Sağnak Taşırlar

Though multicore systems are ubiquitous, parallel programming models for these systems are generally not accessible to a wide programmer community. The macro-dataflow model is an attractive stepping stone to implicit parallelism for domain experts who are not the target audience for explicit parallel programming models.

We use Intel's Concurrent Collections (CnC) programming model as a concrete exemplar of the macro-dataflow model in this work. CnC is a high level coordination language that can be implemented on top of lower-level task-parallel frameworks. In this thesis, we study an implementation of CnC, based on Habanero-Java as the underlying task-parallel runtime system. A unique feature of CnC, first-class decoupling of data and control dependences, allows us to experiment with schedulers by taking these data and control dependences into account for better scheduling decisions. Our observations led to the proposal and implementation of a new task-parallel synchronization construct for Habanero-Java, namely *Data-Driven Futures*.

We obtained two kinds of experimental results from our implementation. First, we compare the effectiveness of task scheduling policies for CnC programs. Secondly, we show that data-driven futures not only reduce execution time but also shrink memory footprint. In summary, this thesis shows a macro-dataflow programming model can deliver productivity and performance on modern multicore processors.

Acknowledgments

I would like to start with thanking my committee for taking time from their busy schedules to listen to my defense, read my thesis and provide feedback. I would also like to single out my academic adviser, Professor Vivek Sarkar, whose patience and effort in the sisyphean task of advising me proved the labor was instead herculean. His initial suggestion for me to pursue this area, introducing me to the community and providing opportunities where the questions arise that is addressed this work, he deserves my deepest gratitude and the most credit for this work. Secondly, my committee member, Dr. Zoran Budimlić have been filling the position that I have imposed on him to be my secondary adviser and provided me with hours of his time discussing technical details under his quite generous open door policy. Professor Keith D. Cooper have been an influence for me throughout my graduate studies as I have listened to his classes, worked as his teaching assistant and in personal communication. I appreciate Professor Lin Zhong's help he provided as a committee member.

I have been lucky to work with the tightly-knit Habanero research group that has been supporting me with constant stream of ideas, technical intellectual curiosity and moral support. I would like to acknowledge the rigorous and prompt technical help Vincent Cavé has provided, without whom nothing would have finished on time. I would also like to thank other group members, Dr. Jisheng Zhao, Dr. Philippe Charles, Dragoş Sbîrlea and Raghavan Raman.

Being an intern in Intel for the Concurrent Collections team has provided me insight to address this work. I would like to thank Dr. Geoff P. Lowney, Dr. Mark Hampton, Dr. Kathleen Knobe and Dr. Ryan R. Newton for their help, support and ongoing collaborative spirit. Kathleen Knobe and Ryan R. Newton have influences

all over this work and Ryan's challenging of the tabular data structure use was the kindling of chapter 4.

I have used applications and benchmarks from collaborators that has helped me tremendously in evaluating my thesis. I would like to thank Aparna Chandramowlishwaran from Georgia Tech whose implementation of Cholesky factorization has been the first real application to show Concurrent Collections' worth, that is also used as an influence in our Cholesky factorization implementation. Fellow Rice graduate student David Peixotto has provided the implementation for the porting of the Black-Scholes benchmark from the PARSEC benchmark suite. UCLA graduate student Yu-Ting Chen has provided the Rician Denoising benchmark written in Matlab, which has been the basis for our implementation. Fellow Habanero group member, Alina Sbîrlea has provided the implementation for the porting of the Heart Wall Tracking application from the Rodinia benchmark suite.

We are grateful to the generosity of NSF for funding the Center of Domain-Specific Computing through the NSF Expeditions and Intel for funding our research.

Finally, I should acknowledge all my friends and family without whom I would not have been able to survive graduate studies. This work is a dedicated to them.

Contents

Illustrations

Chapter 1

Introduction

1.1 Motivation

In the last decade, not only have we observed almost all academic publications and talks beginning with references to Moore's law and how the frequencies have plateaued but also ubiquitous adoption of multicore architectures. This has been a paradigm shift foreseen, nevertheless not well prepared on the programming languages and software engineering fronts. Preliminary remedies were provided by the distributed computing and functional languages communities, yet these approaches have not gained traction as the mainstream programmer and the legacy code-base have had shared memory and imperative programming as axioms.

Parallel programming models have traded conciseness for performance as computer science community is the driving force for these models as 'supply side parallelism'. As multicore adoption brought parallelism down to earth, the trend is to provide more accessible parallelism to a wider audience. Native thread libraries provided an API for users to write parallel programs at the level of hardware threads and thereby exposed details far lower level than necessary. By imposing this inherently nondeterministic construct, it burdens users by pruning the nondeterminacy that is intractable to achieve exhaustively and counterintuitive [1]. OpenMP standard provided a higher level abstraction for the same approach by adopting fork-join model, though have not been the best match for fine grain or unstructured parallelism, which

is of more relevance to a wider community. Data parallelism using array programming languages, vectorizing compilers, data parallel languages like NESL and the recent adoption of GPGPU have been providing expressibility or performance but not both at the same time and have restricted application on programs with intricate dependences. Dynamic light-weight task parallelism, as adopted in Cilk, Intel Threading Building Blocks and Habanero-Java, is the modern trend to alleviate most of the problems mentioned above. Users can now express parallelism in finer granularity without being exposed to low level details and without trading performance for expressibility. This model is also applicable to problems with complicated dependences. Even though this model is a leap in the right direction, it has not attained access to a non-computer-scientist audience. As the discussion deepens in classifying computation as a new pillar of science, our goal is to provide a model with which non-computer-scientists, 'demand side parallelism', can express their applications without performance penalties.

We have embraced macro-dataflow coordination as our parallel programming model since it allows us all the expressiveness and more of models mentioned above. It also provides us the safety nets such as determinism, deadlock freedom and race freedom, yet it still is accessible to a domain expert. Additionally, it breaks the assumption adopted by imperative parallel languages that data dependence is to be satisfied before control dependence, therefore exposing more parallelism. Our goal in this thesis is to provide schedulers for an efficient macro-dataflow model used as a coordination language exposing all the inherent parallelism and map it efficiently to a multicore system with a dynamic light-weight task-parallel runtime. For the scope of this thesis and the embodiment of our ideas, we have chosen Concurrent Collections as the macro-dataflow model where Habanero-Java has provided the underlying

task-parallel runtime.

1.2 Contributions

On this thesis, we

- provide explanations of scheduling algorithms for Concurrent Collections
- introduce Data-Driven Futures as a new language construct for synchronization and item implement Data-Driven Futures for a work-sharing runtime
- implement a data-driven runtime with Data-Driven Future support
- compare various scheduling algorithms and data-driven runtime with Data-Driven Future support both theoretically in our discussions and empirically by implementing various benchmarks and observing effects on execution times and scaling.

1.3 Organization

In Chapter 2, we will provide background as a foundation for our work. The following chapter, Chapter 3, will talk about our design choices, algorithms and the language constructs we used for scheduling CnC. Chapter 4 introduces data-driven futures and a data-driven scheduling mechanism built to support this construct. We will look into some benchmarks, on Chapter 5 to observe how various schedulers perform. Chapter 6 will cover relevant work and where our works fits with respect to previously proposed concepts. Lastly, on Chapter 7 we deduce conclusions based on the empirical evidence and our experience and lay out a plan for follow-up research to this work.

Chapter 2

Background

As our work advocates for a new parallel programming model and we have briefly touched alternative models on our introduction, we will provide a brief explanation for these models here and also cover them on chapter 6 for an elaborated comparison. On chapter 4, we introduce a new variant of a known language construct, i.e. futures, so on section 2.2, we cover what futures are. Following chapters assume an introductory understanding of Concurrent Collections programming model and Habanero-Java task-parallel runtime, both of which we introduce below. Finally, underlying scheduling algorithms of a task-parallel runtime are covered in addition to our Habanero-Java coverage because of their relevance to this work.

2.1 Parallel Programming Models

A parallel programming model acts as a framework to map parallel programming languages to parallelism supporting hardware. Parallel programming languages being an integral part of this framework and most relevant part for this work, we constrain our introduction to the parallel programming languages aspect. Though, we will introduce runtime discussions that fit our work on section 2.5.

By no means the following list is exhaustive, despite it is representative of a sufficiently big subset. We abstained from addressing parallel programming models for distributed memory systems for brevity and relevance.

Parallel programming models are not single dimensional when it comes to which model they follow, so the classification below does not necessarily mean they do not fall under different criterion.

2.1.1 Thread based parallelism

Native thread libraries

used almost synonymously with Pthreads, which is a standardization for native thread libraries for a portable interface. This model provides an API to create threads by forking them from a process or another thread. Thread, in this context is finer grain than a process and coarser grain than a task. On shared memory systems, threads share the memory space of the process and can communicate through that space. Synchronization between threads are established through a join operation, which stalls a thread until the thread to be joined arrives at the synchronization point. Synchronization to block acces to critical regions to prevent data races between threads are supported through a mutual exclusion lock.

This model provides the bare bones of parallelism, exposes threads to their full potential and can be deemed the 'assembly language for parallelism'. Therefore one can use Pthreads to implement data parallelism or task parallelism mentioned below.

OpenMP [2]

is relatively higher level to bare threads. Parallelism is expressed through compiler directives which annotates parallel regions embedded in a serial language. Rather than the explicit thread creation and joining observed on Pthreads, OpenMP implicitly forks slave threads at the entrance of a parallel region, to be joined at the region's exit. It provides interfaces to support loop level parallelism, data and task

parallelism. The type of data parallelism is expressed, is provided by the user via scheduling clauses under the restrictions of data sharing clauses which also is provided by the user. Task parallelism is expressed by parallel region constructs on OpenMP 2 and recently OpenMP 3 has introduced a task concept. Regarding synchronization, OpenMP provides collective synchronization support, barriers, in addition to Pthreads. Critical sections are expressed via a clause named after them, rather than the explicit mutual exclusion locks of Pthreads.

2.1.2 Data parallelism

Data parallel languages

exploit SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) type parallelism, which can be best described as vector parallelism. Instructions applied to a data structure, e.g. an array, is applied to a range of data rather than a single datum. These languages require vector registers and vector instructions on the architecture they are compiled to, for performance. Known examples include High Performance Fortran [3] and NESL [4]. High Performance Fortran exposes data parallelism by directives on the sequential code, where NESL is a declarative language with nested data parallelism is the core construct. Additionally, Intel has been working on Array Building Blocks which has been recently released as a beta version.

SPMD

stands for **S**ingle **P**rogram **M**ultiple **D**ata and as the name implies, it defines parallel programs where a single program is applied on multiple data points. The abbreviation standard reminds of Flynn's taxonomy, on which SPMD would fall under MIMD. A common misconception is the equivalence of SPMD and SIMD. SPMD is more general

in the sense, it does not conform to the synchronized execution of the data stream as SIMD does. As mentioned above, both OpenMP and Pthreads can be made to express SPMD, however we will talk about another exemplar, OpenCL/CUDA.

For general purpose computing on graphics processing units (GPGPU), CUDA is proposed by NVidia and OpenCL by a consortium for open standards for the same purpose. A GPU can be alternatively interpreted as a nested abstractions of computation units. In this nesting, the lowest level computation units exploit SIMD parallelism, where the higher levels exploit SPMD parallelism through shared memory.

2.1.3 Dynamic Light-Weight Task Based

model expects the user to express all the inherent parallelism in a given application at a finer grain level than what thread level or SPMD models expect. This finer grain abstraction is called a *task*.

Dynamically at runtime, these models assign tasks to coarser grain threads and therefore achieve the coarsening by the serialization of tasks assigned to a thread. Though this bookkeeping introduces more runtime overhead, it reaps the benefits of more parallelism exposed and portable scalability, free from the underlying architecture.

Languages pursuing this paradigm can be exemplified by Cilk [5], Intel Threading Building Blocks [6], X10 [7], Chapel [8], Fortress and our choice for this work, Habanero-Java [9].

2.1.4 Macro-DataFlow Model

A dataflow architecture keeps track of the data an instruction needs and executes an instruction as data becomes available, where availability of data imposes the execution

order. This paradigm is an alternate to a von-Neumann machine which is the de-facto industry standard. A von-Neumann machine adopts a program counter, which points to an instruction being executed and updated to point to the next instruction, under the assumption that computations are topologically sorted to conform to their data dependences. One can interpret this model by control-flow imposing execution order. Nevertheless, out-of-order execution support for von-Neumann architectures and register renaming introduces dataflow concepts in a restricted manner.

Macro-dataflow model takes dataflow paradigm to a coarser grain level by describing dataflow not on an instruction but on a much coarser (*task*) level. Tasks are scheduled on this model conforming to the partial ordering defined by the macro-dataflow graph, which is the inherent data dependence graph in between the tasks it is composed of.

Concurrent Collections, to be discussed in more detail in section 2.4, is an exemplar of a macro-dataflow parallel programming model. It expects the user to express the macro-dataflow graph of computations and maps this model to underlying task-parallel runtimes. Therefore it is used as a coordination language that separates the communication (macro-dataflow between the computations) and the computation.

2.2 Futures

Futures bind references to values to those values' computations. This binding is a contract that the reference is resolved through this computation before or during the value needs to be read. Therefore the evaluation of the computation for a value is not imminent at binding time like assignments on eager evaluation languages. Parallelism can be achieved by this delay in resolution of a value, as the binding time can be interpreted as a fork in control-flow and the read as a synchronization point.

Introduction of futures can be traced back to [10] and implementations have been proposed in MultiLISP [11] and many other languages ever since. We will cover the differences in semantics and implementations of futures in section 6.

2.3 Habanero-Java

As our choice for the underlying dynamic light-weight task parallelism language and runtime which we build our CnC implementation on top of, we will introduce Habanero-Java briefly and exclude features that are not directly relevant to this work.

Habanero-Java is an extension to X10 version 1.5, which in turn is an extension to Java, and supports language constructs for portable parallelism, some of which we cover below:

async construct creates a child task which may execute parallel to the parent task. The definition of a child task starts with the keyword `async` and followed by a lexical scope defined by curly braces. A child task has access to the data declared until the parent's lexical scope at the creation point of the child.

finish construct synchronizes all the `asyncs` created within its scope that it is an immediate *parent* scope of. `finish` scopes can be nested and `asyncs` synchronize with their innermost `finish` scopes. This construct also has support for exception handling, as it propagates the exceptions thrown by child `asyncs` that have not been caught. Once an `async` throws an exception, that `async` fails and unwinds without effecting other computations. We will make use of this support on some schedulers covered in section 3.3.

phaser [12] construct provides collective and point-to-point synchronization support. Parallel tasks that have producer-consumer relationships or control dependences register themselves to phaser objects which regulate their execution order. Though we have not used phasers on our work, we will observe the relevance of the phaser concept to the data-driven futures.

We see a sample Habanero-Java code that computes a given Fibonacci number in Figure 2.3.

```
public class Tester {
    public static class BoxInt {
        public BoxInt() { this(0); }
        public BoxInt(int passed) { this.value = passed; }
        public int value;
    }
    public static void fibonacci ( int index, BoxInt result ) {
        if ( index < 2 ) {
            result.value = index;
        } else {
            BoxInt prev = new Tester.BoxInt(0);
            BoxInt prevPrev = new Tester.BoxInt(0);
            finish {
                async fibonacci (index-1, prev);
                async fibonacci (index-2, prevPrev);
            }
            result.value = prev.value + prevPrev.value;
        }
    }
    public static void main(String args[]) {
        Tester.BoxInt result = new Tester.BoxInt(0);
        fibonacci(new Integer(args[0]).intValue(),result);
        System.out.println(result.value);
    }
}
```

Figure 2.1 : Sample Habanero-Java Fibonacci micro benchmark

Habanero-Java is a dynamic task-parallel framework, therefore computations get

dynamically created to be scheduled at runtime. There are two main runtime schedulers to assign tasks to multiple cores, namely work-sharing and work-stealing schedulers. These topics will be covered in section 2.5.

2.4 Concurrent Collections Model

Concurrent Collections (CnC) [13] parallel programming model can be described as a macro-dataflow, coordination language. The model requires explicit declaration of communications, as control and data, between serial kernels at the granularity level of a *task* as a CnC graph. This graph is an extension of a macro-dataflow graph with control-flow edges. By this CnC graph we separate computation apart from the communication and therefore CnC is also a coordination language. A sample CnC program's graphs, both graphical and textual, are represented in Figure 2.2 and 2.3.

Concurrent Collections is provably deterministic and employs dynamic single assignment, consequently it is race-free. Despite the possibility of creating a deadlock like state in CnC by having computations whose data will never be provided (including cyclic data dependent computations) and those tasks will be waited on indefinitely. However, it is easy to pinpoint the error given the determinacy property. We will revisit this discussion below.

Three pillars of the CnC models are *steps*, *items*, control and data *tags*. The static descriptions of these concepts are called *collections*, whence the name Concurrent Collection came. The dynamic instantiations of these *collections* are called *instances*. In Figure 2.2, triangles represent control tag collections, rectangles represent item collections and ellipses represent step collections. In Figure 2.3, item collections are wrapped in square brackets, control tag collections are wrapped in angle brackets and step collections are wrapped in parenthesis.

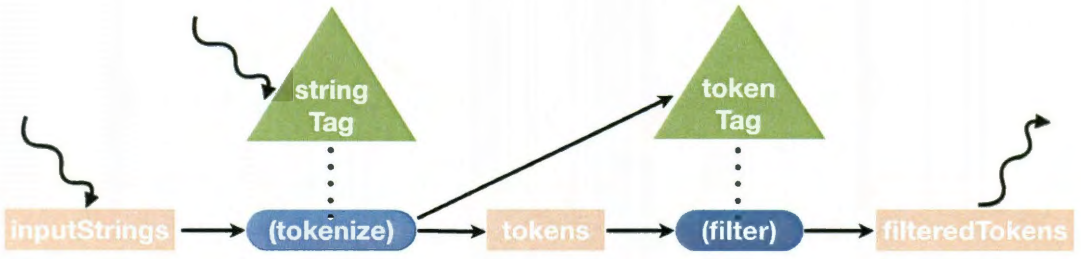

Figure 2.2 : Graphical representation of a sample CnC Graph

Before furthering the explanation, let us address one common confusion which is caused by the overloading of the term *collection*. In statical domain, a *collection* represents what is common to all *instances* that can be generated from that collection. This is the same relationship between a Class and an Object in the object oriented programming paradigm. Nevertheless, all dynamically instantiated entities of a *collection* is maintained by a concurrent data structure during runtime, which also is named a *collection*. We will address the statical descriptions more explicitly as *static collections* and runtime collections remain to be *dynamic collections* to emphasize the differences in between and prevent possible confusions.

```
[String inputStrings];
env -> [inputStrings];
<point stringTag>;
env -> <stringTag>;
<stringTag>::(tokenize);
[String tokens];
<point tokenTag>;
[inputStrings] -> (tokenize) -> [tokens], <tokenTag>;
<tokenTag>::(filter);
[tokens] -> (filter);
[int filteredTokens];
(filter) -> [filteredTokens];
```

Figure 2.3 : Textual representation of a sample CnC Graph

Step is the delineation of computation in CnC. A *static step collection* is a procedure parametrized by *control tag instances*, therefore a *step instance* is an invocation of that procedure with a given *control tag instance.* The CnC *graph* also provides the *prescription* relation, which is a binary relation over *static tag collections* and *static step collections.* This relation describes, which *static step collections* should have invocations with a given *control tag instance* once that *control tag instance* is added to *dynamic control tag collections.* The properties of this binary relation, prescription, is such that its inverse has to be a function. A *static step collection* can be prescribed by at most one *static tag collection*, where a *static tag collection* can prescribe multiple *static step collections.*

The communication into and out of steps is achieved by providing access to the *dynamic item collections* from which a *step instance* is going to read or to which a *step instance* will write. The *item instances* are extracted from and fed into a *dynamic item collection* by a *data tag instance* which is a function of the *control tag instance* that *step instance* is invoked with. A *step instance* may put a *control tag instance* into a *dynamic tag collection* that is has been provided access to, which leads to the invocation of *step instances* of *static step collections* prescribed by that given *static tag collection*, assuming all other conditions (to be covered in scheduling) are met.

Item is a dynamic single assignment value on the CnC domain. A *static item collection* is an abstraction to describe a collection of data statically, which has producer-consumer relationships with *static step collections.* As covered in the previous paragraph, *item instances* are created and consumed by *step instances. Step instances* access an *item instance* by querying a *dynamic item collection* with a *data tag* which is computed from the *control tag* the *step instance* is associated with. The producing

interface for an *item instance* is called a `put` into a *dynamic item collection* and the read interface is called a `get` from a *dynamic item collection.*

Tag serves two purposes in CnC. A *data tag instance* is a key to access a value on a *dynamic item collection*, where a *control tag instance* is a proxy to invoke *step instances* with that *control tag instance* from *static step collections* that have been *prescribed* by the *static control tag collection* of that *control tag instance.*

Given these concepts, a solution to a problem is built by the user providing the CnC *graph* and the kernel computations for the *static step collections*, that are *step* nodes on that graph. Additionally, the user needs to provide a special *step instance*, the *environment*, which initiates the computation by feeding the initial *control tag instances* and the initial *item instances.* Scheduling decisions are made by the underlying runtime schedulers, which will be covered in more depth in chapter 3. For our implementation, we chose Habanero-Java to be the language of the *environment* and *static step collections* and Habanero-Java runtime to be the underlying runtime. The mapping between these two models are achieved in this following manner. The CnC *graph* is translated to create the signatures, abstract Habanero-Java classes, for *static step collections*, which have to be implemented, by extending those abstract classes, by the user to describe the computation that *static step collections* entail. For the *environment* step, the user instantiates a graph object and starts its execution by putting *control tag instances* into some *dynamic tag collections* of that graph. To ensure the synchronization of all the *step instances* that unfolds as the graph is executed, the code introducing the *control tags* into the graph is wrapped within in a *finish* scope. The parallelism is expressed through the implicit wrapping of each *step instance* invocation by an *async.*

One restriction CnC employs is that every get operation within a step implementation precedes a put operation. This is rather a best-practices approach to CnC coding, however it does not solve the deadlock like state mentioned before.

```
step_m(itemCollection_k) {
    ...
    itemCollection_k.put(tag_j,value)
    local=itemCollection_k.get(tag_i)
    ...
}
```

```
step_n(itemCollection_k) {
    ...
    local=itemCollection_k.get(tag_j)
    itemCollection_k.put(tag_i, value)
    ...
}
```

Figure 2.4 : Sample pseudo-code with cyclical data dependences

A pseudo-code snippet that does not conform to the CnC restrictions is depicted above on Figure 2.4. In this code, step instances m and n will wait for each other indefinitely. However, it is possible to transform any code to conform to the all-gets-before-any-put restriction. One possible transformation would split the step implementations in every put operation to make sure every step ends in with a put operation and the rest of the code, the continuation, will be encapsulated in a new step implementation and a new control tag collection will be created to invoke that new step. This new step will be invoked by a put operation for the newly created tag collection for this continuation step. If get and put operations are interleaved this process can go recursively until every step implementation ensures that get operations occur before put operations. CnC does not prevent users from creating a deadlock cycle that is created by a cyclical data dependence, however it should be noted that cyclical data dependence is a design error.

2.5 Runtime Scheduling

On a dynamic light-weight task-parallel runtime, the runtime scheduler needs to assign tasks to threads as they created during execution and maintain load balance with minimal runtime scheduling overhead. Two main subsets for these algorithms are explained below.

2.5.1 Work-Sharing

These schedulers hand out work as new work becomes available to achieve load balance. The centralized work queue approach maintains a global list of ready tasks from which all threads extract new work as they need and once new work becomes available, it is fed into the global task queue. A decentralized work-sharing scheduler, may maintain task queues per thread to represent ready tasks that gets populated by other threads that eagerly share their work once more ready tasks appear.

X10 version 1.5 has an underlying centralized work-sharing runtime implementation which Habanero-Java inherited. Our work and results are based on the work-sharing runtime and scheduling, though work-stealing runtime implementations are listed as future work.

2.5.2 Work-Stealing

These schedulers are inherently decentralized and the sharing of work is not eager, from where the name 'stealing' comes. Every threads task queue maintains a list of ready tasks that are to be executed by that thread. As more tasks gets created, no sharing occurs and they are reserved in the same thread's ready task queue, however once a thread runs out of work, it steals work from other threads' queues to achieve load balance.

There are two alternative policies for work-stealing scheduling administration. Work-first work-stealing, eagerly starts executions of child tasks, leaving the parent task's continuation on the ready task queue. This depth first execution traversal of the task tree is proven to be efficient both in execution time and memory footprint in bounds, given sufficient parallel slackness [5, 14]. Help-first work-stealing policy pushes children tasks created into the ready task queue rather than eagerly starting their execution, which is breadth first traversal of a parent tasks children which exposes a wider computation frontier which helps with lacking slackness in a subset of problems, however the bounding of memory and execution time proof does not hold anymore [15].

Habanero-Java employs both these possible work-stealing policies and also provide an adaptive scheduler [16] that alternates in between as slackness changes during runtime.

Chapter 3

Scheduling of CnC

As dynamic light-weight task based parallel programming models move to the mainstream, runtime scheduling with less overhead, more load balance, hence better performance, proves to be an important research topic. The challenge of engineering a scheduler for such a model is handling varying number and granularity of computations without incurring prohibitive costs during runtime. A taxonomy of popular runtime schedulers has been covered in section 2.5 as the bifurcation of work-sharing and work-stealing.

In section 2.4, we have noted that the Concurrent Collection programming model aims at implicit parallelism through macro-dataflow at the granularity level of tasks. The declarative nature of the model allows us to capture task and data parallelism inherent in any application. The expression of parallelism is attained by the Habanero-Java code compiled from a CnC application, using `async-finish` constructs. Consequently, we execute the parallel code on the Habanero-Java runtime.

Arcs on a dataflow graph represent data being passed from a node to another and for relaxed dataflow models these data can be tagged in order not to delay nodes on their output channel. As we have covered before, we use a hashmap that is accessible by every computation and can be queried with a given tag, to simulate data channels. However, dataflow or data-driven scheduling *fires* computations when their data is ready, nevertheless our model allows computations to be created ahead of time because of the introduction of control-flow as a first level construct.

To simulate dataflow semantics or a data-driven runtime scheduling, we use blocking on uninitialized data as an alternate solution. Additionally, we use delayed asyncs(see section 3.2), and introduce data-driven futures on chapter 4 as our proposed constructs to help us with the mapping of CnC on a task-parallel environment. The following sections 3.3 and 3.4 describe implementations of CnC schedulers in two main subsets, namely eager schedulers and data-driven schedulers.

3.1 Challenges

The parallelism expressed in CnC is not trivially mappable to an `async-finish` computation. The challenge arises from CnC's expression of control dependences between tasks possibly before data dependences are satisfied. The control dependences, on CnC domain, determine *if* a computation is going to be executed but *when* it is going to be executed is determined based on the satisfaction of data dependences or lack thereof during runtime.

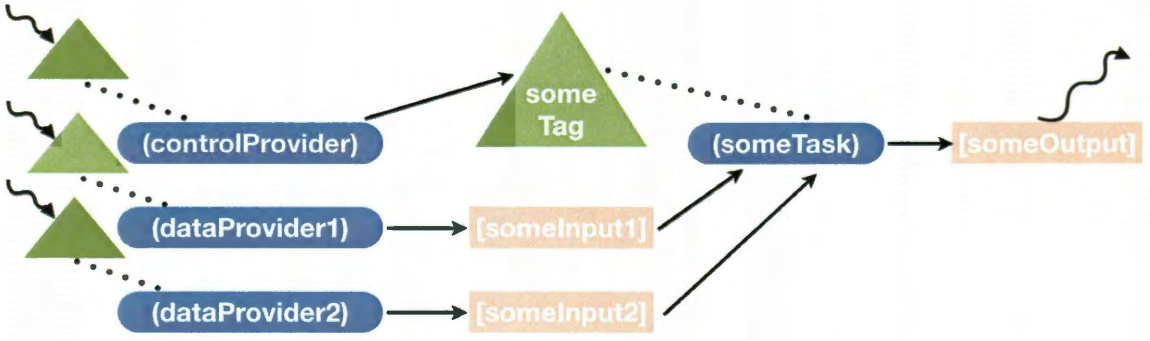

Figure 3.1 : Simplified sample CnC Graph to show the separation of control and data providers

For example, let us look at a simplified sample CnC graph on Figure 3.1. In a given execution trace, let us suppose that a *controlProvider* step instance puts control

tag instance *(i,j)* into the *someTag* control tag collection. This will eventually lead to the execution of a *someTask* step instance. However, there is no guarantee that the item instances from item collections *someInput1* and *someInput2* that the *someTask* step instance will read, would have been provided preceding the `put` of the control tag instance. Therefore the execution of the *someTask* step instance has to delayed at runtime to ensure that the needed data is ready.

In some models, take fork/join or serial languages, *if* and *when* are tightly coupled; if a computation will be executed, it may be executed anytime beginning from when it is told it may be executed. For example, Habanero-Java's `async-finish` style parallelism on a work-stealing runtime with work-first or help-first policy, in which an `async` does not guarantee immediate execution though it does not prohibit imminent execution either. The execution of a task may be delayed until the enclosing `finish` is observed, while all other siblings may be running or finished execution. However, this is still more restrictive than the CnC approach, since in CnC a task may be created and marked logically parallel whether it is ready or not, even possibly preceding the tasks with which it may have data dependences.

Even though the separation of *if* and *when* is not exclusive to CnC, most models conform to the notion that data dependences of a task is satisfied prior to the expression of the control dependence. As most parallel languages are embedded into or implemented on top of serial languages, this imperative, serial semantics of programs percolate up to their parallelized versions. Tasks labeled to be parallel in these languages uses values that are currently in lexical scope, which guarantees that the values are created before the parallel task is forked apart. This property does not hold for CnC, which introduces possible complications to be handled if CnC is to be mapped to a task-parallel language like the ones mentioned above.

3.1.1 Methods and restrictions for CnC

Only restriction CnC imposes on the implementation is that every step computation should perform all `get` operations before any `put` operation as noted in section 2.4. However we need to restrict the step definition even further to achieve better performance.

Let us assume that a step performs a costly computation before performing any `get` operation. The number of executions of that step until the `get` operation is scheduler dependent and can be repeated many times. Therefore to get better performance from out implementations, we have adopted a best-practices-policy of performing all the `get`s in the beginning of a step definition. Hoisting all `get` operations to the beginning of a step definition is trivial under the assumption that the `get` operations are data and control dependence free and executed regardless of any other computation within the step. It may seem restrictive to enforce `get` operations to be control independent however this is as expressive as the default model and this transformation can be done automatically.

Let us assume that a `get` operation is control dependent to the results of many other computations, even including other `get` operations results. This hampers the analyzability of the code makes the readiness checking and possible rollbacks more expensive. We instead transform these types of codes into steps definitions beginning with control dependence free `get` operations by expressing the control dependence at the CnC graph level instead of within the step. We duplicate the step code from the first `get` operation point where the one copy has the condition satisfied and the other copy has it unsatisfied. We create these two new kinds of step collections and we also create two tag collections to spawn these steps. Additionally, we change the original step definition to evaluate the condition and put a tag into the appropriate

tag collection based on the condition and finish execution.

Following the best-practices or employing this transformation lets us statically analyze the data-dependence graph, cuts readiness checking and rollback costs.

3.2 Delayed Asyncs

Delayed asyncs, introduced in [17], are guarded execution constructs, that extends Habanero-Java `asyncs` to be susceptible to a boolean flag which denotes whether the computation is ready or not.

The syntax adopted for delayed asyncs is `async (`⟨*booleanExpr*⟩`)` ⟨*Expr*⟩. The scheduling and therefore the execution of ⟨*Expr*⟩ is delayed until boolean expression ⟨*booleanExpr*⟩ evaluates to true. The challenge here is to decide when and how frequently to evaluate the boolean expression to promote delayed asyncs to full-fledged `asyncs`. The answer is dependent on the given underlying runtime scheduler.

As discussed in 2.5, our work-sharing scheduler maintains a global task queue which is populated by `asyncs` scheduled, though not yet run. Prior work provided augmentation to support delayed asyncs is to treat them as if they are proper `asyncs` when extracted from the work pool. They evaluate their boolean expressions to check if they are ready to run and if they are, they get executed. However if they are not, they get re-queued and go to the back of the queue. This may seem as a potential for overhead, since at worst case (a post order traversal of the dependence graph of step instances), it is possible to cause $O(N^2)$ total re-queue operations readiness checking operations, where N is the number of all possible step instances, even if all step instances were to finish executions instantaneously. The reason for this cost can be expressed as follows. Since the queue may be the postorder traversal of the readiness graph the very first available task will be at the end of the queue therefore it is needed

to evaluate all N queued tasks to find the first available task. This whole queue checking will continue as the queue size shrinks which would lead to $\sum_{i=0}^{N}$ readiness checking which amounts to $O(N^2)$ readiness checking and re-queueing operations.

Under real-world circumstances, an execution of a step instance is not instantaneous. Consequently, it is possible that all threads looking for tasks to execute, may dequeue and re-queue the whole task queue over and over again until some ongoing task execution finishes, which is preventing its dependents from running. One may classify this behavior as busy-waiting on a queue. However, our observations led us to believe this is never the case and even if the program dumps all possible step instances to the work queue, this pathological case has not been created with real world benchmarks that we implemented. Applications of this construct will be covered in section 3.4 as Delayed Async scheduler is based on support mentioned.

Delayed async support is implemented differently on work-stealing runtime since there is no global queue to enqueue tasks to delay execution once they are dequeued prematurely. So, we have a queue to store all delayed asyncs for a `finish` scope. Once a worker walks into the end of a `finish` scope, rather than picking up the continuation, the worker traverses the delayed async queue and promotes all ready delayed asyncs to normal `asyncs`. This relaxation loops until the delayed async queue is empty, when the worker picks the continuation just like the default case.

3.3 Eager Scheduling

We have mentioned previously how CnC is inherently different than task-parallel languages by pointing out that the data dependences may not have been satisfied at the point where the control dependence is expressed. Therefore, one possible scheduling mechanism is to assume that the data for any task is ready and treat

tasks whose control dependence is satisfied as ready for execution. However, we need to make sure that the execution of eagerly scheduled tasks are either delayed or blocked until their data are ready. If we also allow eager execution, we should unroll the computations that started without their data being ready. Follows are the schedulers that employ these policies.

3.3.1 Blocking Scheduler

This policy makes sure you conform to the data dependences by using `gets` with blocking semantics. A blocking `get` waits on an item that it is called to consume until that data is ready. We use the `wait-notify` mechanism provided by Java to implement this functionality.

If a task blocks on a failed `get`, so will the coarse grain thread which holds that light-weight task. At worst case, we may block all the threads since the tasks on the computation frontier may not be enabling any formerly blocked threads. We remedied this condition by augmenting our work-sharing scheduler to have varying number of threads on its thread pool as threads block to avoid deadlock. However one may point out that there will be more than number of threads initially designated on the pool when blocked threads get woken up. This not only introduces context switching cost but also increases the demand for memory if too many threads get blocked. The same augmentation could have been applied for a work-stealing scheduler with even more complications so we did not provide an implementation for that case.

Blocking schedulers for work-sharing runtime have been proposed as a naïve initial step and therefore a baseline for other schedulers. We will visit performance results in detail in chapter 7.

Coarse Grain Blocking

In this particular policy, every item collection has a single monitor to synchronize access to the whole collection. Therefore a failed `get` will wait on the single monitor. Despite the simplicity, the downside to this approach is that every `get` will be using the same monitor. So even if a `put` with data tag (k) will be waking up threads that are waiting on other other item instances with different tags. All unnecessarily waken up threads need to be eagerly run again to either read the value if it has been put or lock the same monitor again. As one would expect because of multiple failures and the contention on the monitor, this scheduler is expected to under perform, which will be compared on the results section.

Fine Grain Blocking

We proposed finer grain monitor synchronization to alleviate the problems mentioned in the previous paragraph associated with a coarse grain locking approach. For this policy, every item instance has its own monitor. Therefore once a `get` fails on an item instance with data tag (k), only a `put` with data tag (k) will notify those threads and only those threads. This approach reduces contention on locks by giving each item a monitor and also threads are only waken up when the data they slept on gets ready.

3.3.2 Data-Driven Rollback & Replay

Eager blocking schedulers, as explored on the previous subsection, suffer from blocking coarse grain threads just to block light weight tasks. In order to mitigate this problem, one needs to disassociate tasks from threads once they become a hazard for the whole thread. Data-Driven Rollback and Replay policy scheduling is built based on this premise.

Like any other eager scheduling policy, it is assumed that a task is ready for execution even if the data dependences are not satisfied. During the execution of a task, if a `get` is performed on a data anticipated to be ready but is not, an exception is thrown to unwind all the computation that has been done. Since CnC computations are functional with respect to their inputs and are side-effect free, this rollback and possible rerun in the future is legal. Preceding the exception thrown, the computation that executed the failed `get`, notes itself as a closure to the data it failed to read. If more tasks are to fail reading the same data, they stack up their closures to the same entry. When the data is made ready by a corresponding `put` call, all closures recorded to have failed are scheduled to be rerun.

The explanation above, ensures that a task can not run to completion before its data is ready due to the exception for reading uninitialized data. Additionally, a coarse grain thread does not block or gets unwound, since the tasks do not block and the exception is caught by the Habanero-Java runtime just to be ignored, however unwinding the task it originated from. This approach observes the fine grain data dependences of a task during runtime by stumbling on the data items a task performs `get` on, hence we named the policy data-driven rollback and replay.

One handicap for this policy though, is the possibility of multiple failures before execution. If a task features multiple `gets`, gets scheduled, picked to be run and fail to succeed in the first `get`, the closure for that task will survive in the queue for the data it failed on. When that data is `put` by some task, it will get reinstated. On the second run, the first `get` will succeed, although it is possible to fail on the second `get` attempt and get queued waiting on the second data. On worst case, it is possible to reinstate a task for the number of `gets` it performs and incur the cost of creating a task, scheduling it and unwinding the computation it may have done by then by and

exception. These observations laid the groundwork for our Data-Driven Future work, which will be covered in greater detail on chapter 4.

3.4 Data-Driven Scheduling

Even though all legal schedules of a program will delay any task until all required data is ready, eager schedulers achieve this goal by assuming the data would be there during execution optimistically and handling error cases if the assumption fails. In contrast, data-driven schedulers delay the execution by checking whether the data is ready or not without execution. In these schemes, computations delegate the data dependence checking to additional constructs which fails on behalf of the computation. We propose two schedulers complying with this policy. Firstly, the Delayed Async scheduler with the help of the language construct we covered in section 3.2 where it gets its name from. Secondly, we propose pure data-driven scheduling that is influenced by dataflow execution model with the help of the language construct, data-driven futures, to be covered in chapter 4.

3.4.1 Delayed Async Scheduler

Given a guarded execution construct like delayed asyncs, this policy converts all computations into guarded computations with guards being the data dependences. Even though the item instance level dependences can be expressed on the CnC graph ahead of compile time or can be deduced by the compiler at compiler time, for simplicity we assume that the guard is provided by the programmer. Since we provide the programmers with abstract classes to provide implementations to based on their declaration of computations, we also provide a readiness function stub to be implemented. This boolean readiness function has the same parameters that the computation function,

where the expectations from the programmer are to check the presence of the data tags for items the computation would consume.

Once all tasks are created as delayed asyncs, where the guards validity represents the satisfaction of data dependences, the runtime scheduler with the delayed async support discussed in section 3.2 makes sure a task is run only once and after all its data is provided. As we discussed in that section, not throwing away tasks that should not have run in the first place or not blocking coarse grain threads translates into better performance. Yet again, there also are inherent downsides to this approach. Delayed async evaluation, which is the delegation of the safety of a computation, has to be repeated rather than the whole computation. This seems as an advantage given that the guard evaluation is cheaper than the computation, though it may be a bottleneck and turn into busy waiting in unlikely cases.

Chapter 4

Data-Driven Futures

In this chapter, we introduce a new synchronization object, namely Data-Driven Futures, to keep track of inter-task dependences based solely on data and to provide support for a dataflow like scheduling policy. We have implemented data-driven futures on Habanero-Java using a new data-driven runtime scheduler. Section 4.1 introduces data-driven futures. In section 4.2, we describe a data-driven runtime scheduler we implemented that can be used to support data-driven futures and includes a discussion on a blocking runtime scheduler for the sake of completeness. Finally section 4.3 elaborates how we implement this concept.

4.1 Introduction

A data-driven future (DDF) encapsulates a value and acts as its proxy. Nor the value may have been computed and neither the task to resolve that value may yet have been created when a DDF is instantiated. Data-driven futures provide support for the regulation of accesses of tasks that have a consuming relationship with that value and the producing task. Below is a suggested interface for this language construct:

get is the interface for accessing the result of a data-driven future. If the DDF has already been provided a value via a `put`, described below, a `get` delivers that value. However if the producer task has not yet been created or its execution not finished at the time of the `get` invocation, that `get` fails. The definition of failure is

dependent on the underlying runtime scheduler. On a data-driven runtime scheduler, the expected action is an unrecoverable error, as that `get` should have never been executed. In contrast, on a runtime scheduler with blocking support, the task from which the `get` is invoked should synchronize with the producer task and wait for it to be created and its execution finished. The latter case scales data-driven futures down to mere futures, so one may support that functionality. Further discussion on runtime scheduling aspects of DDFs are covered in section 4.2.

put provides the functionality for the resolution of a data-driven future. Every DDF has a unique computation that would resolve the value associated, which we call the producer. Once the producer initializes the data field of a DDF, it needs to wake all consumers that have either not been spawned or blocked preceding that `put`. As DDF is a reference to a value and not a variable, only one producer may set the value and any other attempt at setting the value, should be an unrecoverable error independent of the underlying runtime.

Creation is mere creation of a reference object that points to nothing. Both producer task and consumer tasks may have handles for this object, where the producer task resolves the reference to an actual value via a `put` and consumer tasks dereferencing it via a `get`. It is possible to provide the computation or even the value during creation, though that blurs the distinction from mere futures and as mentioned above, we will abstain from that discussion.

Registration describes the association relation between a DDF and the tasks consuming it. A task, designated to consume DDFs, is created by registering itself to all the DDFs it may read. These registrations help regulate the spawning of tasks for

execution on a data-driven runtime and helps resolving which task to synchronize on a blocking runtime. Further discussion on runtime scheduling aspects of DDFs are covered in section 4.2.

```
// Create two DDFs
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish { // begin parallel region
    async left.put(leftBuilder());//Task1
    async right.put(rightBuilder());//Task2
    async await (left) leftReader(left);//Task3
    async await (right) rightReader(right);//Task4
    async await (left, right)  bothReader(left, right); //Task5
}  // end parallel region
```

Figure 4.1 : A Habanero-Java code snippet with Data-Driven Futures

The sample code snippet in Figure 4.1 shows five logically parallel tasks and how they are synchronized through DDFs. Initially two DDFs are created as containers for data items `left` and `right`. Then a synchronization scope is started via a `finish`, which harbors five logically parallel tasks, annotated with `asyncs` as in Habanero-Java and X10. The registration declaration of which DDFs a task should read, is expressed by an `await` clause in the `async`. The tasks suffixed `Reader`, are passed references to perform a `get` on the DDF instances that they receive.

For instance, the fifth task registers itself on both `left` and `right` DDFs, which declares a data dependence from the first two `asyncs` that are the producers for those DDFs. Regardless of the underlying scheduler, the first two `asyncs` are guaranteed to execute before the fifth `async`. This ability for a task to wait on two (or more) DDFs is unique to our DDF model, and was not supported by past work which will be covered in section 6.

4.2 Runtime Scheduling with Data-Driven Futures

4.2.1 Motivation

Data dependence edges as first level constructs

Concurrent Collections programming model builds a coarse grain task graph statically, where the fine grain task graph is exposed as the computation unfolds during runtime. However, the exposed task graph, which we are calling frontier from here on, is not necessarily just *enabled* tasks. This is a direct conclusion from the case we built on the previous chapter based on how control dependences are not sufficient to deduce readiness as they do not encapsulate data dependences. As we have covered in our discussion on eager schedulers, the computation frontier for those schedulers consists only of ready task nodes though it may mislabel tasks, that have not satiated their data dependences, as ready. Therefore our computation frontier is unnecessarily larger which increases book-keeping costs and incur scheduling costs for tasks that should not have been scheduled. In order to maintain an execution frontier, we need to maintain what is currently available, what will become available and what has completed.

Default programming convention is to describe a task dependency graph linearly by providing a valid topological sorting of these tasks. For example compilers use dependency analysis to reverse engineer the intended task graph from the linear code. As this linear code is traversed, every encountered computation is ready and schedulable since the description of the program followed the aforementioned convention. This is not much different for parallel programming languages either. Since there are multiple flows of control, it is possibly unsafe to read effects of another flow of control preceding a synchronization point, so it is safe to assume all parallelly executed tasks

are independent of each other. Therefore a correct, race-free parallel program also conforms to the same convention. There are exceptions to this rule, e.g. one can describe a parallel program where there are control dependences between the parallel tasks. In order to guarantee safety, a synchronization between those steps have to be adopted. For example Habanero-Java phaser[12] construct is a remedy for this example.

Data-Driven Futures serve the data dependency equivalent of a synchronization object for parallel tasks with data dependences in between.

Arbitrary task graph construction

The nested fork/join model, widely adopted by many current parallel programming models, restrains the edges of the task graph to *unify* control and data dependences. The source of the dependency edge in those graphs (parent) provides the child not only with control but also with data through its lexical scope. However, the parent task that creates a downstream parallel computation, does not necessarily create the data consumed by that computation.

For example, looking back at Figure 4.1, we see that the dependence graph between those five tasks can not be described by standard nested fork/join without constraining parallelism. The *data* dependency graph between the tasks in Figure 4.1 can be described by the left side of Figure 4.2. This graph can not be created with nested fork/joins operations. An alternate solution in a fork/join model would be to hoist $task_1$ and $task_2$ to the parent task, thereby creates an implicit barrier between producers and consumers. That approach would result in a dependency graph represented on right side of Figure 4.2, which has less parallelism than left side of the same figure. Additionally, the lack of a construct like DDFs burdens the programmer

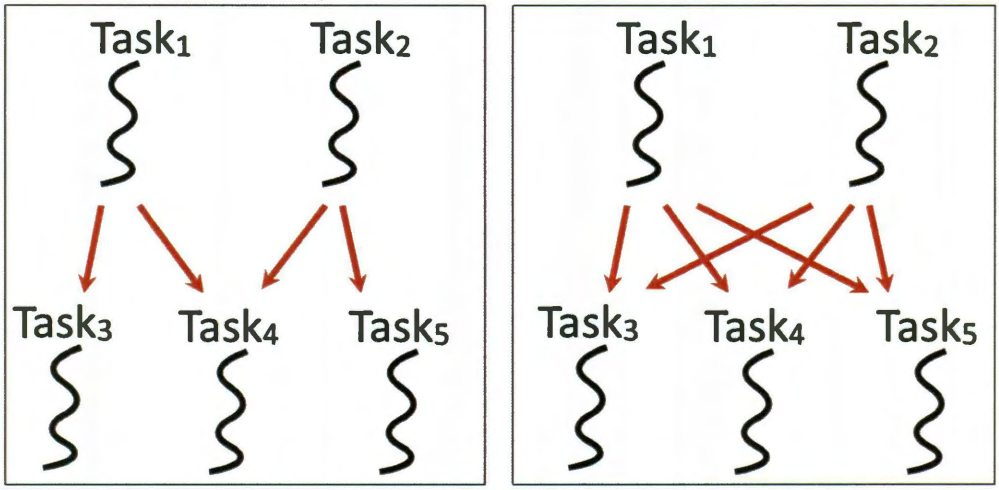

Figure 4.2 : Data dependency graph of Figure 4.1 (left) and the *unified* dependency graph of the fork/join equivalent of the same program

in thinking about creating fork/join structures to satisfy data dependences.

4.2.2 Methods and restrictions for tasks awaiting DDFs

We have adopted restrictions for our coding principles using DDFs parallel to the discussion mentioned in section 3.1.1. When a task declares to read a set of DDFs through its waiting list, we expect that all these reads are data and control dependence free. If not, one can adopt the same transformation described in section 3.1.1 but with no need for tag collections as they are irrelevant for this scope. Hence, we can hoist the data and control dependent `get` operations out of a step by splitting the step into the data and control dependence free subparts. Additionally, any computation deemed not data dependent on the Data-Driven Futures in the waiting list does not belong into a task that awaits them and therefore they should be hoisted out, too. We have assumed these restrictions for implementation and as we have shown before these restrictions are as powerful and expressive as the non-restricted counterpart and

can be transformed into and out of.

4.2.3 A Data-Driven Runtime Scheduler supporting Data-Driven Futures

A data-driven runtime supporting DDFs would follow asynchronous dataflow semantics. This runtime assumes that all tasks that consume data-driven futures would register themselves to what they are consuming. As in dataflow scheduling, the tasks that are to be executed are not spawned or invoked but rather consume the data they declare to consume once that data becomes available. Therefore, it is the availability of the data that runs the scheduling, not explicit scheduling requests. Consequently when a DDF becomes resolved, every task registered as its consumer would become ready, given that is the only DDF that task is waiting for. If there are multiple DDFs for a task to consume, one can think of the latest arriving DDF as the enabling one.

As we have briefly touched on the effect of runtime scheduling to data-driven future semantics on section 4.1 and also elaborated on the paragraph above, it is possible to provide a data-driven runtime scheduling with these design choices:

- Registration of tasks to data-driven futures they consume
- No explicit invocation of a task that consumes DDFs
- DDF signal consumer tasks as they become resolved
- Invocation of a task when all DDFs, which that task registered to consume, have signaled

We have implemented this runtime scheduler, whose implementation details are covered in section 4.3 and performance results compared to alternative CnC schedulers can be observed on chapter 5.

4.2.4 A Blocking Runtime Scheduler supporting Data-Driven Futures

As in the blocking versus waiting design choice on future constructs on other domains or as in CnC scheduling policies mentioned in chapter 3, one may choose to implement the support for data-driven futures by blocking threads from which a `get` is performed on an uninitialized DDF until the corresponding `put` occurs. Additionally, the design choices can be furthered in handling the case where the computation leading to the corresponding `put` is known. One possible choice is to lazily wait for that task to execute at schedulers sake and another is to enforce execution of that computation.

As we have covered the possible implications of blocking and the overhead associated with it deduced from our experiences implementing blocking schedulers for CnC, we have not provided an implementation for this variant. We have concluded that a data-driven runtime scheduler is a better fit to show the benefits of data-driven futures, therefore only data-driven runtime scheduler variant is implemented.

4.3 Implementation

Our implementation for data-driven futures and the data-driven runtime scheduling supporting this construct is based on Habanero-Java and the work-sharing runtime of Habanero-Java. Accordingly, within this sections scope it is appropriate to read a *task* as a Habanero-Java `async`, an *object* as a Habanero-Java object and a *list* as linked list implementation by Java utilities library.

Data-Driven Futures are objects that hold a single-assignment value and a linked list of tasks registered as consumers of this value waiting for that value to be assigned. In general, the value, will be assigned at runtime by a producer task. Since the value held within a DDF is single-assignment, any attempt to reassign the value results in

an exception.

Each task holds a list of Data-Driven Futures it is designated to consume. This list is populated during the creation of a task at runtime where the compiler introduces the initialization code. Readiness of a task can then be checked any time by a traversal over the list of DDFs. A consumption ready DDF would be one with the data field already assigned. Additionally, since the readiness of a single DDF is monotonically increasing (from uninitialized to assigned and never to be reassigned or to be uninitialized again), so is the readiness of the whole list of DDFs. Once a DDF is found to be ready, we can stop checking for its readiness. Every DDF list can retain a state of where the ready part of that list ends using an iterator and the sublist being waited on starts, in order not to unnecessarily check ready parts that can not be made not ready.

We can see a partial sample configuration snapshot in Figure 4.3. This figure shows the data dependence relationships between tasks A, B and C through the DDFs α, β and δ. Here are some conclusions we can derive from this snapshot. First of all, we know $task_A$ will consume data items in DDF_α and DDF_β, where $task_B$ consumes data items in DDF_β and DDF_δ. The task designated as the producer for DDF_β is $task_C$. Some producers have already provided the values for DDF_α and DDF_δ. From the upper left corner of the figure, we can see that a DDF has a list of tasks which are its consumers and tasks have a list of DDFs they consume. At the time of this snapshot, $task_A$ has already passed over DDF_α since the value has been produced and is not waited on anymore. However on $task_B$s case, even though DDF_δ is ready, the task is not aware of that fact yet as it is waiting on DDF_β. In this scenario, let us assume that the very next action is the assignment of the value DDF_β is synchronizing. The assignment of that value will induce the traversal of

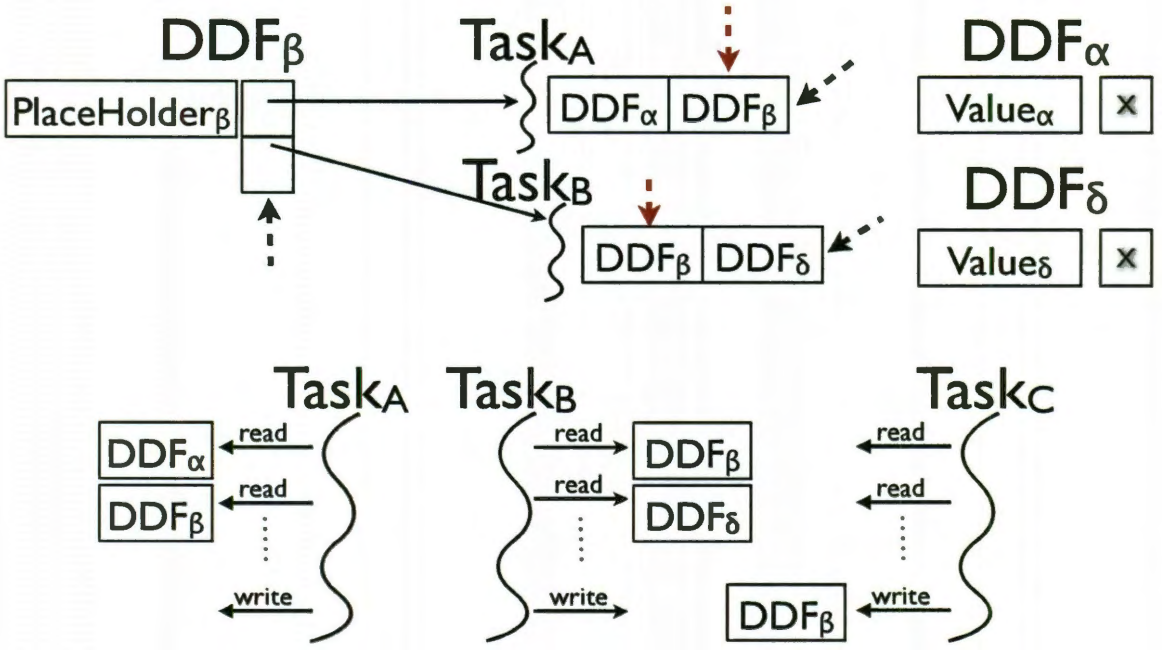

Figure 4.3 : Snapshot of a subset of Data-Driven Futures and tasks during runtime

the designated consumer task list. On every task, the wait frontier, shown by a red dotted arrow, is iterated, which is an asynchronous way of moving every task to its next phase. For example, $task_A$ will observe the end of the list that will cause its eventual execution as it is deemed ready where $task_B$s next phase is to check the readiness of DDF_δ, which will succeed and cause $task_B$s execution.

We conform to the semantics for data-driven futures described above with the following API we imposed on our Habanero-Java implementation.

get is implemented as it is covered in section 4.1 given the underlying scheduler is data-driven runtime scheduling. If an incorrectly implemented program attempts to access a data-driven future that has not been resolved, an exception is thrown. As

the data that is being referenced by a DDF never changes, there is no synchronization on this method for performance concerns.

put also conforms to our discussions above for a data-driven runtime scheduling support for a data-driven future. This method is synchronized with readiness checking method in order to ensure isolation property, in the sense that not a half initialization would be observed. A DDF being initialized maybe polled during the initialization by another thread for readiness. Once the initialization is complete, `put` exits the critical region to advance the iterators indicating where tasks have been waiting for all the tasks registered themselves as consumers. Any task created from then on, will observe the DDF to be ready, therefore consumer tasks succeeding a `put` do not have to be advanced.

Creation instantiates an DDF object that is a placeholder for the single-assignment value that DDF is a reference to and sets the list of tasks registered to be consumers to an empty list. It is also possible to initialize the data field during creation, which can be described as the providing of the computation for that DDF, where that computation is a constant expression. We do not allow for the data-driven future to be a computation, i.e. like a Habanero-Java future, for simplicity.

Registration is currently provided by the user creating the task. We require tasks to declare on what DDFs they perform `get` on. The syntax we proposed for this declaration is as follows: `async await` $(ddf_a, ddf_b, \cdots)$ $\{\langle Expr \rangle\}$. For example, going back to Figure 4.3, declaration of $task_A$ would be `async await` $(ddf_\alpha,\ ddf_\beta)$ $\{\ \cdots local_a = ddf_\alpha.get(); local_b = ddf_\beta.get() \cdots\ \}$

The changes on a data-driven future consuming task are as follows:

Initialization now also initializes the list of data-driven futures that task consumes by extracting the list mentioned in the registration paragraph above. The state of the task representing on which data drive future it is currently waiting on, the iterator, is set to the beginning of the list and subsequently the advance method is called to skip over the data-driven futures which have already been resolved prior to the creation of this task. This advance method is synchronized and this is the reason why: Let us say the very first call to advance, to skip over the data-driven futures already resolved, finds the first unresolved DDF and inserted itself as a waiting task to that DDF. Before the advance called by the initialization can return as it has inserted itself to a waiting list, a put on that object happens and calls advance on that task. Now the waiting frontier is advanced but the initial advance call returned failure and no other advance will be called on that task as it is not registered as a consumer at any other thread.

Invocation is delegated to the list of DDFs readiness condition, rather than the eager enlisting of a task into a task queue. In this version of a task, running to end of the list of DDFs and therefore observing the conformance of data dependences triggers the inclusion of the task to the global task queue.

Chapter 5

Results

5.1 Methodology

We test our work and present performance results on the two following machines.

Xeon machine has a quad-core Intel E7730 processor running at 2.4 GHz. Each processor has two pairs of cores, where each pair shares a L2 cache of size 3MBs. Amount of total main memory for this machine is 32 GBs. For our tests on this machine we set the number of workers to be 16, that is one worker per core.

Niagara has a Sun UltraSPARC T2 microprocessor that has 8 cores and supports concurrent execution of 8 threads per core. There is only one L2 cache of size 4MBs to be shared between all these cores. For our tests on this machine we set the number of workers to be 64 to attach workers to all hardware threads.

For both machines above, we use Sun Hotspot JDK 1.6 JVM and since these are 32-bit versions we cap the memory usage to 4GBs. In our Cholesky factorization benchmark that uses optimized vendor libraries, we used Intel Math Kernel Library(MKL) version 10.2.3.029. For Heart Wall Tracking benchmark, we used GNU C compiler version 4.1.2 to build the underlying C code that is called from the Habanero-Java via native interface.

All the tests presented below are either the minimum running times or the mean running times of 30 runs of a benchmark from a single invocation to ameliorate

inconsistencies of a execution times on a time sharing system, clean and dirty cache impacts and effects of just-in-time compilation. Heart Wall Tracking is the exception to the 30 test runs as deterministically crashes after 13 runs, which is the limit we have used to conclude minimum and average execution times. We have been influenced by [18] in our choice for 30 runs of a program in an invocation and confidence intervals for the mean.

We have dubbed the single threaded execution of our parallel benchmarks as 'serial' on our charts. One may argue that it is not the most conventional use of the term and also fails to address the overhead of parallelism. This is an intentional choice to set the scope of this work and we plan to provide overhead analysis on future work.

5.2 Benchmarks

5.2.1 Cholesky Factorization

For a given symmetric, positive definite square matrix, Cholesky factorization calculate two factors whose multiplication equals that matrix. The factors are lower and upper triangular matrices, where the upper triangular matrix is the transpose of the lower triangular matrix. This may be interpreted as a variant of square root.

This particular algorithm is a well known dense matrix linear algebra kernel that is used frequently in the literature as it is an efficient way to find solutions to systems of linear equations, part of linear least squares calculations for linear regression, eigensolvers and many other applications. This algorithm allows various parallelism opportunities like pipelining, loop parallelism, task parallelism and nested data parallelism. However exploiting all these possibilities for parallelism is not a trivial task

and Concurrent Collections proved to be best solution[19].

Performance Results

This section provides a summary of our performance results on our Cholesky factorization implementations. We have implemented tiled Cholesky factorization using Concurrent Collections, where steps are either pure Habanero-Java code or calls to Intel MKL. Additionally, we present data-driven future versions of both the pure Habanero-Java and MKL using versions, which does not use Concurrent Collections. We have not covered the tiling granularity aspect on this section as it is orthogonal to our discussion.

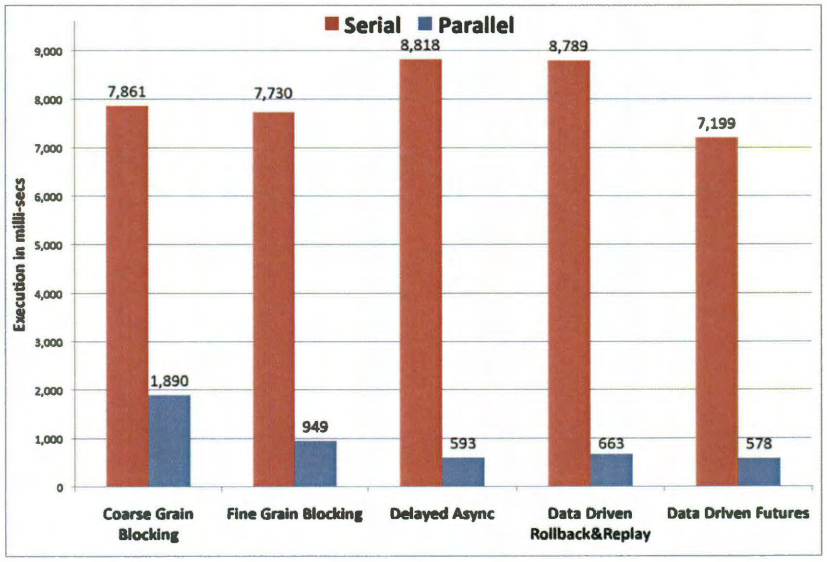

Figure 5.1 : Minimum execution times of 30 runs of single threaded and 16-threaded executions for blocked Cholesky factorization CnC application with Habanero-Java steps on Xeon with input matrix size 2000 × 2000 and with tile size 125 × 125

On Figure 5.1, we can observe speedups of 14.8, 13.2, 12.4 for data-driven rollback & replay, delayed async and data-driven future schedulers, respectively, on a Xeon with 16 cores. As mentioned in the introduction of schedulers, the aforementioned

schedulers do not suffer from blocking bottleneck that proves to be an inhibitor to scaling on this benchmark. Since all possible work that is to be done is provided as the program initiates, there are going to be many tasks that fail as their data would not be ready at that point.

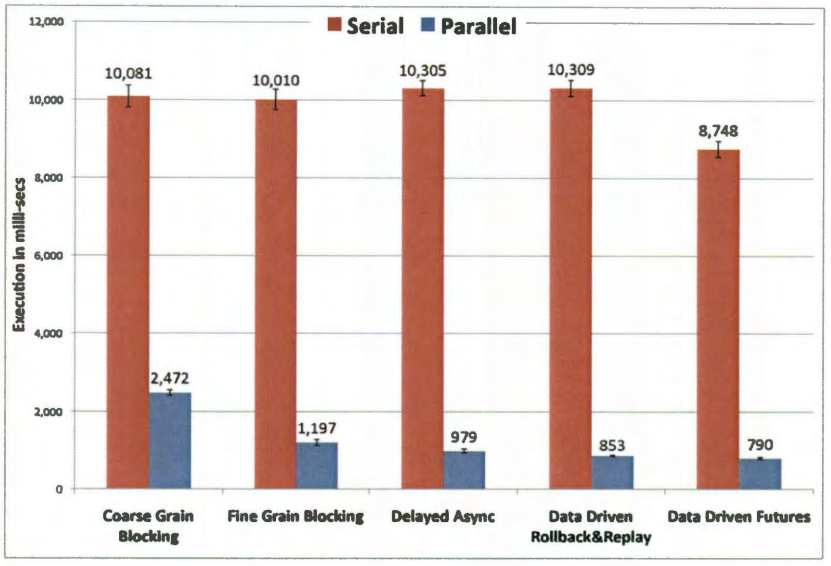

Figure 5.2 : Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Cholesky factorization CnC application with Habanero-Java steps on Xeon with input matrix size 2000 $\times$ 2000 and with tile size 125 $\times$ 125

During our explanation of data-driven future scheduling, we compared it to data-driven rollback & replay scheduling and pointed out it has less overhead as it does not throw exceptions to unwind the computation that failed a `get` and it does not replay a computation the number of `gets` times that computation has. So the numbers fit with our expectations. Likely, we presented data-driven rollback & replay scheduling as an improvement to delayed async scheduling, since the readiness of a computation is not checked by continuously polling if the data needed by that computation is ready. However the numbers on the chart show that the fastest execution time for

delayed async scheduling is lower than the data-driven rollback & replay one. This is a statistical outlier as the average running times on Figure 5.2 show the picture we are expecting to see, where delayed async scheduler takes more time than data-driven rollback & replay scheduler and that scheduler takes more time than data-driven future scheduling on average.

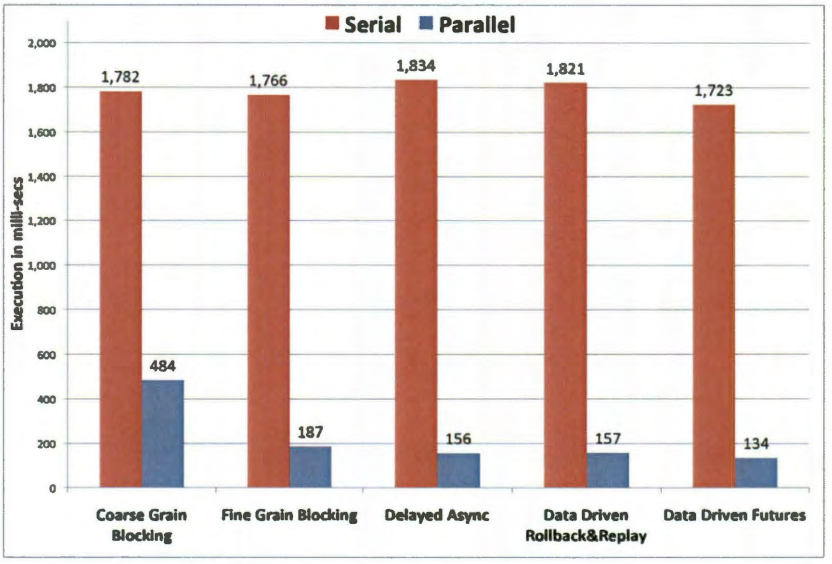

Figure 5.3 : Minimum execution times of 30 runs of single and 16-threaded executions for blocked Cholesky factorization CnC application with Habanero-Java and Intel MKL steps on Xeon with input matrix size 2000 $\times$ 2000 and with tile size 125 $\times$ 125

Figure 5.3 and figure 5.4 feature a benchmark which is a different use case where Concurrent Collections is used more explicitly as a coordination language where computation instances are library calls. We used Intel MKL to calculate the computationally expensive part of the problem and used CnC to regulate the data dependences under a parallel execution of these library calls. We see faster execution with respect to pure Habanero-Java computation on both the serial and the parallel execution times. The speedups for the minimum execution times are 11.7, 11.5, 12.8 for data-driven rollback & replay, delayed async and data-driven future schedulers,

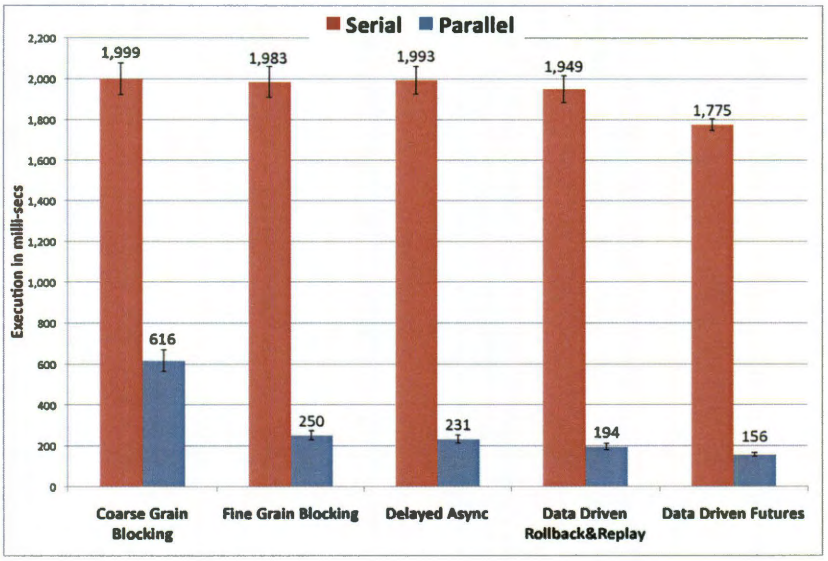

Figure 5.4 : Average execution times and 90% confidence interval of 30 runs of single and 16-threaded executions for blocked Cholesky factorization CnC application with Habanero-Java and Intel MKL steps on Xeon with input matrix size 2000 $\times$ 2000 and with tile size 125 $\times$ 125

respectively, on a Xeon with 16 cores where the speedups for the average execution times are 8.63, 10.06, 11.41. The discussion on the Cholesky implementation results above also applies here as scheduling is not dependent on the underlying computation language.

The results for Cholesky factorization benchmark on Niagara, as figure 5.5 and figure 5.6 show, follows the same patterns we have observed on the previous figures featuring the execution times on a Xeon machine. One interesting observation is that the benchmark achieves speedups over 18 for all schedulers on the minimum execution time case and over 15 for average execution time case where this machine has eight floating operator units and this benchmark is computation bound as a dense linear algebra kernel.

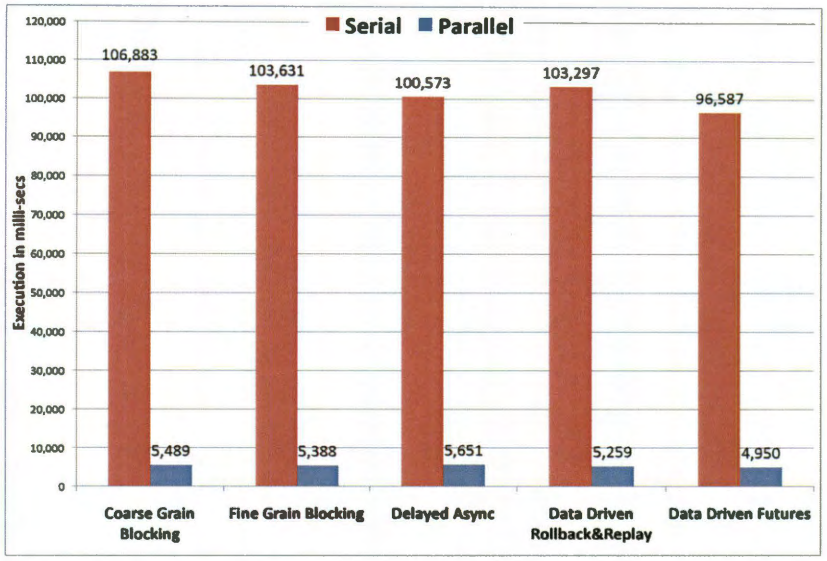

Figure 5.5 : Minimum execution times of 30 runs of single threaded and 64-thread executions for blocked Cholesky factorization CnC application with Habanero-Java steps on Niagara with input matrix size 2000 × 2000 and with tile size 125 × 125

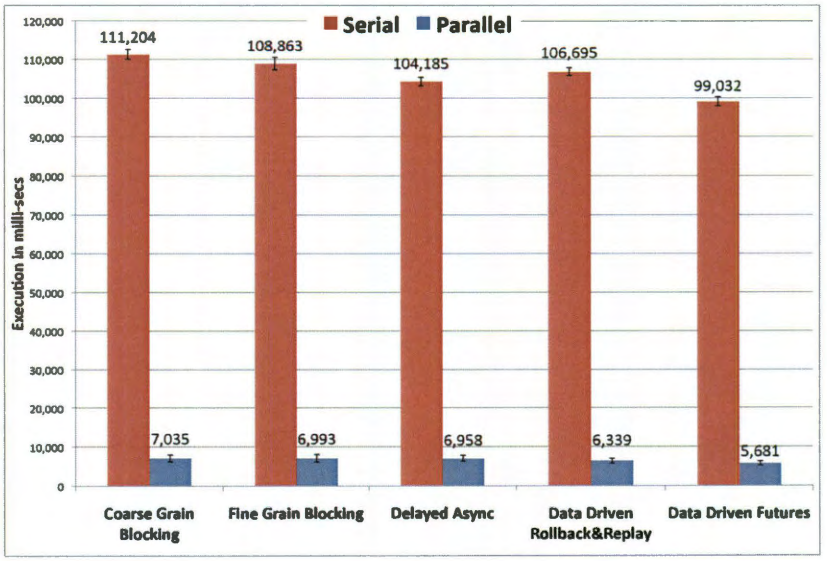

Figure 5.6 : Average execution times and 90% confidence interval of 30 runs of single threaded and 64-thread executions for blocked Cholesky factorization CnC application with Habanero-Java steps on Niagara with input matrix size 2000 × 2000 and with tile size 125 × 125

5.2.2 Black-Scholes

This benchmark calculates the option pricing with Black-Scholes model partial differential equations. This model, for which the Nobel Prize in Economics is awarded in 1997, applies equations with various parameters for input data points and calculates resulting data points and therefore does not have any data dependence besides the user provided input and the programs output. One can look at this problem as a pipeline with a single stage or a streaming problem.

We chose this benchmark as a representative for embarrassingly data parallel problems and it also is featured in PARSEC [20] benchmark suite. Since there are no data dependences of interest in this benchmark, it is a good indicator of overhead.

Performance Results

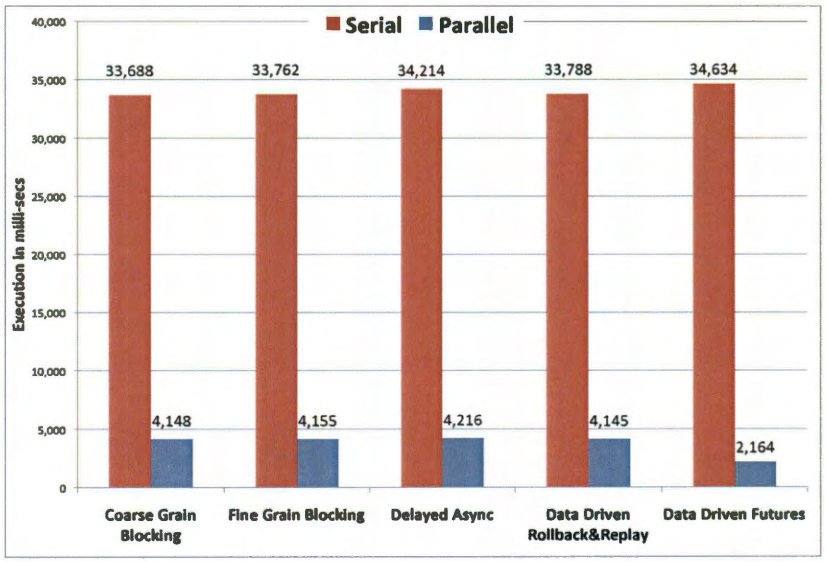

Figure 5.7 : Minimum execution times of 30 runs of single threaded and 16-threaded executions for blocked Black-Scholes CnC application with Habanero-Java steps on Xeon with input size 1,000,000 and with tile size 62500

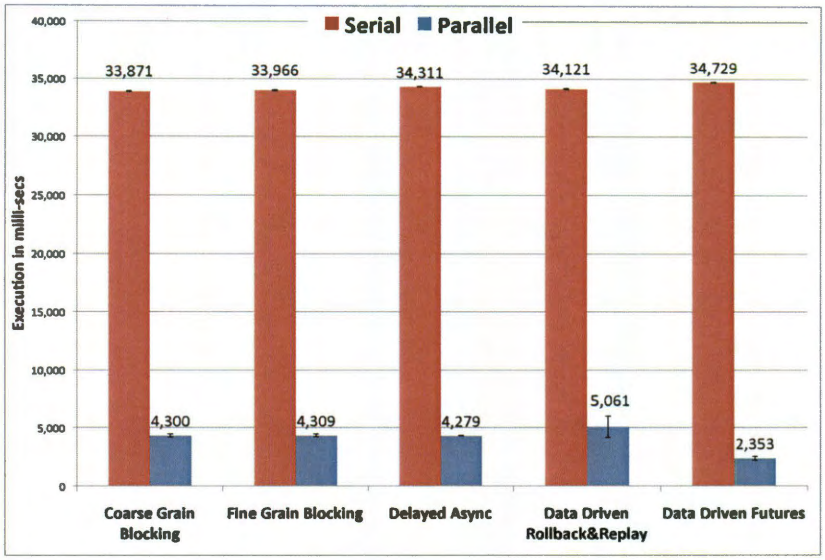

Figure 5.8 : Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Black-Scholes CnC application with Habanero-Java steps on Xeon with input size 1,000,000 and with tile size 62500

As this benchmark can be parallelized perfectly and no tasks have data dependences in between, what we see from figure 5.7 and figure 5.8 are the effects of overhead associated with the bookkeeping. On this Xeon machine every scheduler topped its speedup at 8 where data-driven futures scaled linearly on Xeon by a 16 speedup.

For the Niagara tests, we have set the tile size to allow 64-way parallelism by keeping the problem size the same. Therefore it should be noted that we are trying to achieve strong scaling for this benchmark. As can be observed from figure 5.9 and figure 5.10, we achieved speedups of 27.23, 27.98, 29.87, 30.31 and 40.71 on the schedulers represented in the figure left to right for the minimum execution times and speedups of 26.78, 33.62, 34.61, 35.09 and 38.29 for the average execution time case. Possible explanations for this phenomenon can be explained with memory bandwidth, thread level parallelism implementation of Niagara and the overhead we may have introduced by our model. However in any case, as in others data-driven

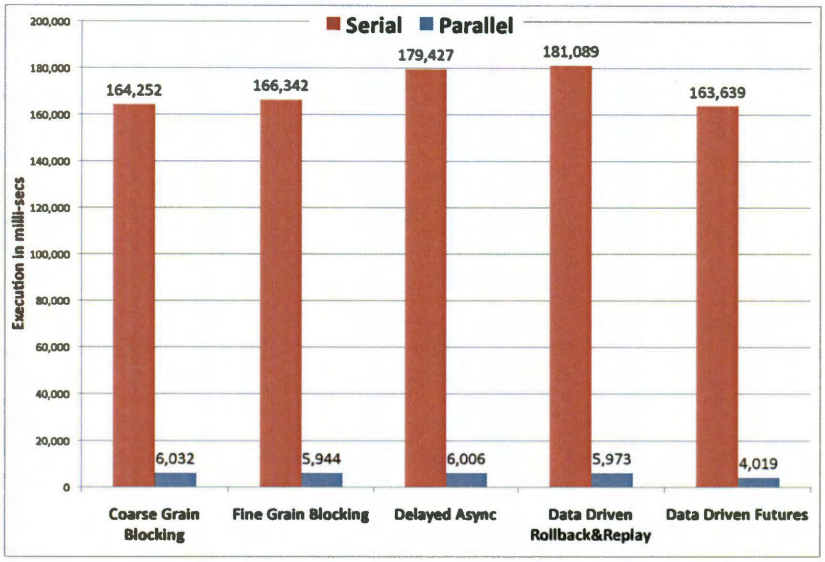

Figure 5.9 : Minimum execution times of 30 runs of single threaded and 64-thread executions for blocked Black-Scholes CnC application with Habanero-Java steps on Niagara with input size 1,000,000 and with tile size 15625

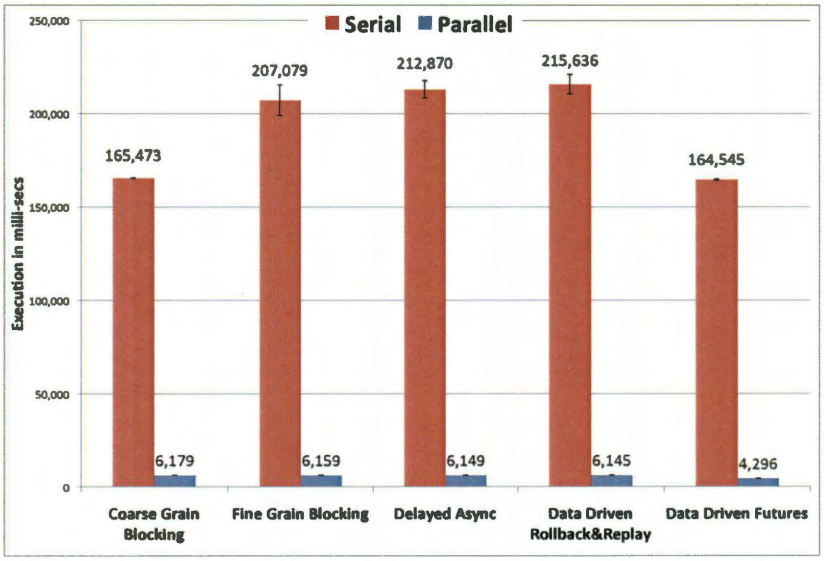

Figure 5.10 : Average execution times and 90% confidence interval of 30 runs of single threaded and 64-thread executions for blocked Black-Scholes CnC application with Habanero-Java steps on Niagara with input size 1,000,000 and with tile size 15625

future scheduling tops the performance chart.

5.2.3 Rician Denoising

This stage of medical imaging pipeline, as the name implies removes noise from images. The name Rician comes from the Rician noise model. The computation is a fixed point, stencil computation where the amount of data touched and created is vast. That is a particularly hard challenge for a single assignment model like Concurrent Collections. As we do not know how many iterations leads to convergence, we can not exploit parallelism across iteration. Additionally, intermediate steps are not relevant to the results once they have been used but because of the dynamic single assignment semantics of CnC, they are kept which prohibits the execution of large problem sets. Our experiences with this benchmark and the observations on memory footprint contributed to our motivation for a data-driven execution model.

Performance Results

Figure 5.11 and figure 5.12 show us results where delayed async scheduler managed to outperform data-driven future scheduling by 1% on minimum execution time and 0.5% on average execution time. However data-driven future scheduling beats any other scheduler by far on single threaded execution times. Besides there is more parallelism available, the highest speedup for these schedulers is 9 on this 16 core machine

This benchmark as mentioned in the lead in, is a fixed point computation that converges after an unknown number of iterations. Correct usage of data-driven futures cut the lifetime of values to their bare minimum by breaking the tabular nature of CnC item collections, which makes this problem solvable with realistic memory

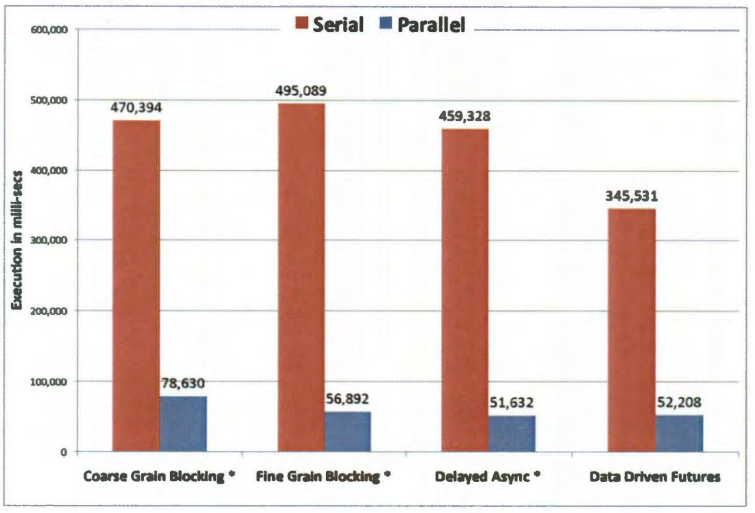

Figure 5.11 : Minimum execution times of 30 runs of single threaded and 16-threaded executions for blocked Rician Denoising CnC application with Habanero-Java steps on Xeon with input image size 2937 × 3872 pixels and with tile size 267 × 484 (Scheduling algorithms with * required explicit memory management by the programmer to avoid running out of memory)

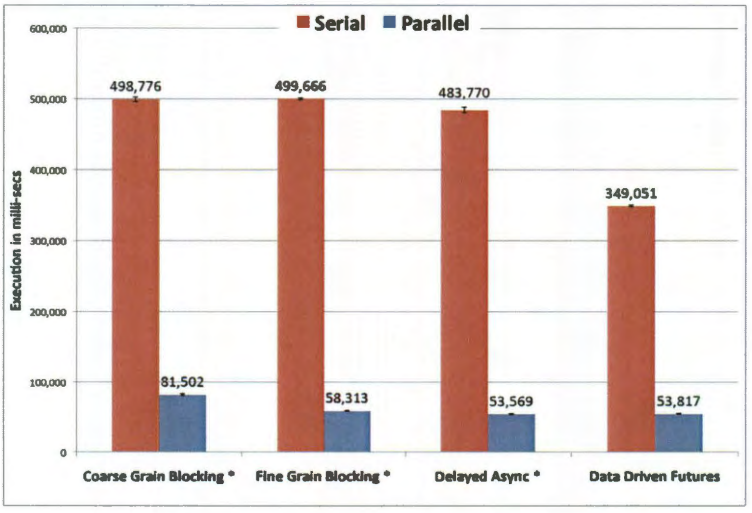

Figure 5.12 : Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Rician Denoising CnC application with Habanero-Java steps on Xeon with input image size 2937 × 3872 pixels and with tile size 267 × 484 (Scheduling algorithms with * required explicit memory management by the programmer to avoid running out of memory)

requirements. We managed to collect partial CnC numbers, by using explicit memory deallocation after every iteration which does not naturally belong in the CnC interface. Therefore, it also should be noted that data-driven future scheduling not only provides more performance by less overhead but also exposes unnecessary referencing of data which single assignment blurs and reduces memory footprint by letting garbage collection deallocate unaccessible memory.

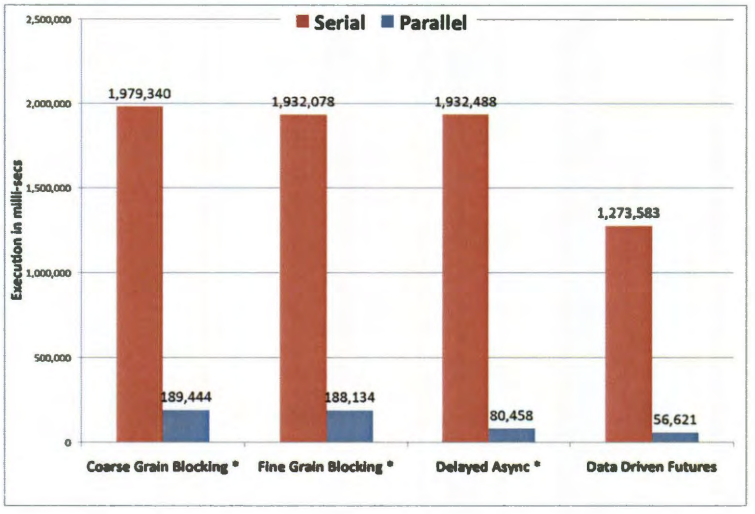

Figure 5.13 : Minimum execution times of 30 runs of single threaded and 64-thread executions for blocked Rician Denoising CnC application with Habanero-Java steps on Niagara with input image size 2937 × 3872 pixels and with tile size 267 × 484 (Scheduling algorithms with * required explicit memory management by the programmer to avoid running out of memory)

Figure 5.13 and figure 5.14 shows us the results of the explicitly memory managing CnC scheduling policies and data-driven future scheduling on Niagara. On this machine delayed async do not surpass data-driven future version as in the Xeon version.

Speedups observed on this machine have been around 22 for delayed async scheduling and data-driven future scheduling. Given the tile sizes and the problem size for

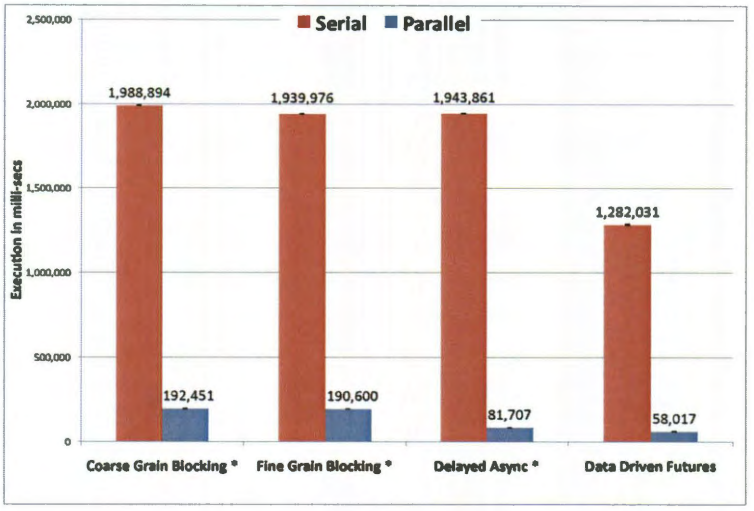

Figure 5.14 : Average execution times and 90% confidence interval of 30 runs of single threaded and 64-thread executions for blocked Rician Denoising CnC application with Habanero-Java steps on Niagara with input image size 2937 × 3872 pixels and with tile size 267 × 484 (Scheduling algorithms with * required explicit memory management by the programmer to avoid running out of memory)

the charts above, there are 11 × 8 tiles. As this benchmark is a 5-stencil computation, the parallelism is available on the diagonal wavefront. On an 11 × 8 tile problem, the wavefront is of size 19 that is smaller than the speedup achieved.

5.2.4 Heart Wall Tracking

This medical imaging application keeps track of a hearts motion on a given set of images as video. Since every image is dependent on the previous one there is no parallelism exploitation across images. However points in an image are free from each other, which is taken advantage of in this example. We have acquired this benchmark from the Rodinia benchmark suite [21] and applied CnC as a coordination language to their serial kernels.

The heart wall tracking benchmark applies various tasks to the individually in-

dependent points on an image and the dependency graph of these tasks are not series-parallel, therefore expressing this synchronization is not trivial in most parallel programming models.

As indicated in methodology, this benchmark is the exception to the 30 test runs. It deterministically crashes after 13 runs, from which we calculate minimum and average execution times.

Performance Results

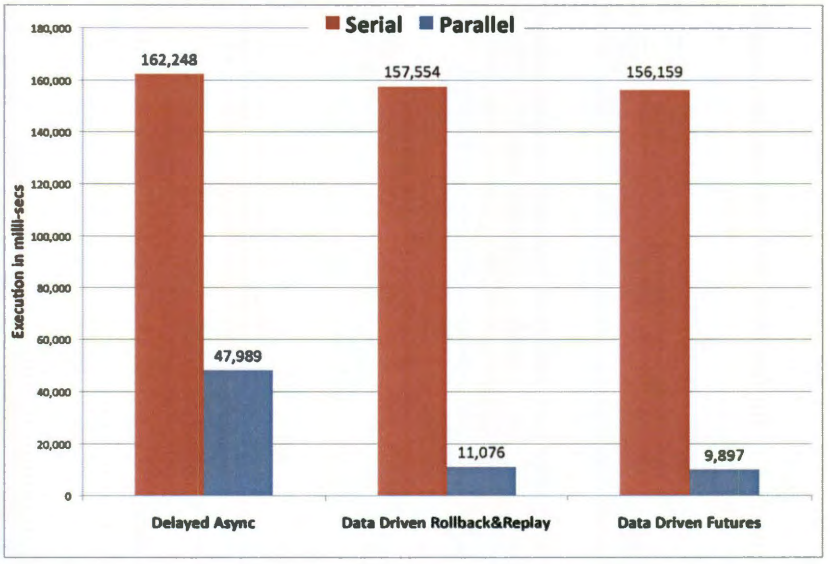

Figure 5.15 : Minimum execution times of 13 runs of single threaded and 16-threaded executions for Heart Wall Tracking CnC application with C steps on Xeon with 104 frames

The intricate nature of the dependency graph of this benchmark's tasks, makes it likely to have plenty of failed scheduling attempts if an eager scheduler is adopted. Data-driven rollback & replay scheduler does not suffer drastically, as failure registers the task to be revived to the data it failed on and continues execution on another task. However as discussed in the previous chapter, blocking schedulers cause the

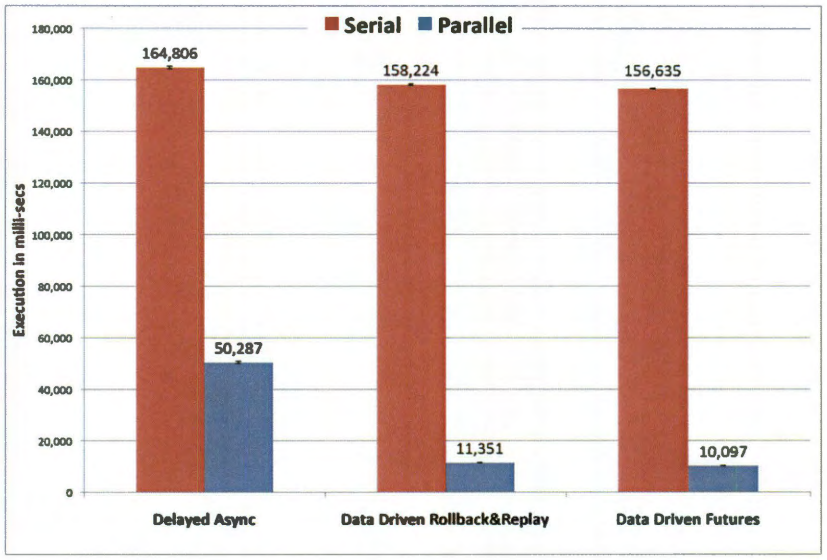

Figure 5.16 : Average execution times and 90% confidence interval of 13 runs of single threaded and 16-threaded executions for Heart Wall Tracking CnC application with C steps on Xeon with 104 frames

whole thread maintaining the task to be blocked. On a problem with an intricate dependency graph and with fine grain computation size, many threads will block, which is what happened in this benchmark. Blocking schedulers run out of possible number of threads that can be created and run out of memory, which is why there are no numbers depicting their performance on figure 5.15.

Data-driven futures not only outperform other schedulers both on single threaded and multi-threaded execution but also have better scalability as the speedup for minimum execution time and average execution time are both over 15.5 on a 16-core Xeon machine.

Chapter 6

Related Work

6.1 Parallel Programming Models

Native threading libraries, like Pthreads, suffer from the lack of abstraction it provides as discussed in chapter 2. Besides the exposed technical details leading to boiler plate code, parallelism exposed in this model does not provide the same scalability once ported. OpenMP can scale up and down though it still is verbose and needs expertise to achieve performance. It is also easy to write code that suffers from race conditions and false sharing. Additionally, OpenMP is not a good fit if the dependency graph of the application is more complicated than a plain fork/join graph. Data parallel models to take advantage of SIMD parallelization are restricted to a restricted subset of parallelism not a big subset of benchmarks would fit and SIMD machines have not been adopted widely by the market with the exception of SSE instruction support and partly GPGPUs which are hard to code for as they expose all hardware details. SPMD models concern the users with granularity, does not allow the exposure of all the inherent parallelism and, as mentioned for the cases of OpenMP and GPGPU languages, can not describe benchmarks with intricate dependences with ease.

Most parallel programming models, like OpenMP, CUDA, Cilk, Intel TBB, .Net Task Parallel library are nondeterministic and imperative in nature. X10, which is the influence for Habanero-Java, has a declarative subset and High Performance Fortran supports declarative data distribution. Concurrent Collections parallel programming

model disburdens its users from creating race conditions by its declarative nature and dynamic single-assignment property. CnC is also provably deterministic which aids debuggability and composability and makes tracing a parallel code seamless by freeing the user from thinking about intractable number of possible interleavings [22]. StreamIt is a declarative language however it is suited for streaming applications where CnC has a broader base. Dataflow languages have lost traction, though one contemporary example would be LabVIEW [23].

We have identified CnC as a macro-dataflow model, though we do not know of mainstream macro-dataflow models to compare CnC to. According to [24], CnC conforms to most commonalities of dataflow languages and extends them by bringing control-flow to a first level construct, which remedies the unintuitive recursion or iteration space declarations needed in dataflow languages. The addition of control-flow as a first level construct to a dataflow language may seem counterintuitive, however this is achieved through *tags* which resembles the solutions proposed by dataflow languages. One can interpret a control tag just as another type and instance of data input.

The tag space of a CnC execution may remind of the tuplespaces of Linda [25], which is an influence for the CnC model and the reason why we labeled CnC as a coordination language [26]. One key difference is that a read access in Linda is destructive and it is not single assignment which leads to nondeterminism. Additionally, Linda uses the tuplespace to hide communication mostly in distributed sense of communication, where we kept the scope of CnC, at least for this work, to a shared memory model.

The Nabbit [27] library extension to Cilk++ provides the user support to declare arbitrary task dependences to create task graphs that can not be created with nested

fork/join operations. However, it still has only one type of dependence, which is control dependence. The unification of data and control dependences cause unnecessary hoisting of producer tasks which extends life times of values and hampers parallelism, as we discussed in section 4.2.1.

6.2 Futures

Futures (also as promises, eventuals) have been proposed [10] and an early implementation of this construct can be seen in MultiLISP [11]. Many other languages have proposed variations and implementations ever since, including Habanero-Java that we will cover below.

Initially futures implied eager semantics, where creation of a future meant the binding of a computation to a reference that initiates the evaluation. The value is either resolved until it is referenced or forced to resolve by an explicit invocation or by blocking. Additionally, the E language and some others have used futures as an abstraction to hide communication in a distributed setting, influenced by [28]. Alternatively, a future can be evaluated lazily like thunks [29]. Some languages conform to a different naming standard about futures and imposes a distinction between producer and consumers, for example the latest C++0x standard draft uses a *promise* as an interface for providing the resolution of an asynchronous computation where a *future* is the consuming interface.

Habanero-Java features a future construct, that can be best explained as an `async` that returns a value. Since an `async` does not execute immediately at the point declaration and ensures it will complete by the end of a `finish` scope, it can be used as a building block for a future implementation with augmentation. That augmentation is the `get` or `force` interface to futures that defines a read action and ensures

completion of execution before a `finish` scope's end.

One inherent difference between data-driven futures and futures is the availability of the computation at creation time. A future is a reference to a result of a computation but you need to have the computation that you are making a reference of at binding time. However DDFs are references that can be assigned only once after they have been created. Therefore one may describe a complicated set of producer-consumer relationships with more ease compared to basic futures. Even though, it would be illegal to have cyclic producer-consumer relationships, expecting the programmer to provide a partial order on declaration of futures during implementation to be able to express correct producer-consumer relationships is an unnecessary burden without any benefits on expressiveness. Data-driven futures remedy these complications.

There is no interface to force the resolution of a DDF and the consumer tasks do not get created until the DDF is deemed ready by the producer of that DDF. However, a DDF that is to be passed to its producer tasks will eagerly get resolved, so the DDF can be described as eager resolution semantics once the producer computation binds to it. One construct that can also be labeled a type of future, I-Vars/M-Vars [30] have much in common with DDFs. Both are place-holders for a single-assignment value and both provide a `get` and `put` interface and therefore separate the binding of the resolver computation and the declaration of the synchronization object. However, our understanding is that DDFs allow a task to await for an unordered set of any size to be waited on where it is not trivial to achieve the same semantics with I-Vars. M-Vars are the sibling of I-Vars that does not conform to the single assignment semantics and we do not allow multiple assignments to a DDF.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

We addressed the accessibility of known parallel programming models and how a macro-dataflow parallel programming model helps a broader domain by implicit parallelism. This model expects parallel applications to be expressed by laying down kernel computations, their interactions with respect to data and the causality relations. These high level abstractions, almost at the level of software engineering design concepts, prevent the presupposed seriality imposed by imperative languages and parallel programming models built on top of them. Our exemplar CnC, as a coordination language, decouples computation and communication and thereby takes advantage of the performance of underlying imperative parallel languages for kernel computations but delegates the communication aspect to the runtime, which this work addresses.

We challenge the notion that data dependences between tasks are regulated by control dependences. Mainstream parallel programming models ensure data dependence compliance by making the parent task provide the data needed to the child tasks. This requires the programmer to represent tasks that are not actually control dependent to be control dependent and impose a topological sort of data dependent tasks during programming. On an application with a complicated control and data dependence graph, this proves to be an onerous and error-prone task. We remedy this problem by adopting Concurrent Collections.

As it has been covered in the literature, modern task-parallel programming languages and runtimes provide performance, scalability and load balance. However, as mentioned above breaking assumptions task-parallel frameworks presuppose, we had to bridge this assumption gap by our work by providing schedulers and constructs that ensure safety as task-parallel languages define safety.

We implemented a new construct, Data-Driven Futures, to support arbitrary task graph creation by declaring data dependences between tasks. This new construct extends the expressiveness of futures and provides support for data-driven scheduling decisions when used as a synchronization construct. Additionally, it allows tasks to wait on an arbitrary set of unordered synchronization objects which is novel to this construct.

We conclude by on our empirical results that a data-driven runtime with data-driven future support outperforms data-driven rollback and replay scheduling which in turn outperforms delayed async scheduling that outperforms blocking schedulers. We observe competitive results for both structured and unstructured parallelism baring benchmarks both for absolute running time and scalability. Additionally, data-driven futures restrict the lifetimes of variables to their absolute necessity, which also relinquishes the user from having memory footprint concerns during implementation and still retains the dynamic single assignment abstraction.

We believe this macro-dataflow model implemented on top of a task-parallel runtime increases the abstraction level for application domain users, provides them with safety nets and fills the gap between expressiveness and performance.

7.2 Future Work

7.2.1 Locality aware scheduling with DDFs

As we have covered in background, Habanero-Java also employs an adaptive and locality aware scheduling policy. This scheduler can be used to expose and take advantage of the memory hierarchy which is crucial to performance in today's architectures. If we can integrate the locality aware scheduling underneath our data-driven scheduling, one can assign affinities to data-driven futures and tasks and the research challenges remain to be solved when out of place data or computation access is needed. This may also be used by a tuning expert to constrain the memory footprint of a program.

7.2.2 DDF support for a work-stealing runtime

We introduced work-stealing runtime and schedulers in section 2.5, mentioned their performance and the bounds it provides. We believe it is possible to take advantage of these schedulers using data-driven futures.

We have provided an implementation of data-driven futures for work-sharing runtime which enqueued tasks that become ready to be popped by idle threads looking for work. However because of the decentralized nature of work-stealing scheduler, it is not immediately apparent which threads ready task queue an enabled task needs to go to. One possible answer may be to ship an enabled task to the latest data provider or the control provider if all the data were ready. However this assumes that when a data is provided, it needs to know the context from which it originated. That is likely to increase cost either through bookkeeping or contention. If an enabled task goes to a random thread's ready task queue, then it is possible to suffer performance penalties because of locality.

7.2.3 Compiling CnC for Data-Driven Runtime scheduling

When we introduced CnC and provided artificial examples. It should have been apparent that the dependence relation is between *collections* not *instances*. It is natural to associate a data-driven future object with an instance and we have implemented our DDF equivalent of CnC benchmarks by associating every item instance with a data-driven future. Therefore we need to deduce the dependency relations in instance grain rather than collection level to be able to map CnC programs down to a data-driven future equivalent.

In [31], it has been proposed to use *slicing annotations* to describe the dependences in instance level by annotating what function of the tag is used to access which instances of data through the textual CnC graph specifications. We believe given these annotations, it is possible to describe a CnC application using DDFs automatically. One can describe a CnC item collection as an array of DDFs and it is known through the textual graph what step instances will read which DDF instances. However this is not going to restrict the lifetimes of the DDFs therefore can not be used to restrict the memory footprint. One possible fix is for the compiler to split the DDF array representing the item collection to the innermost scope while it is safe to do so to restrict the lifetimes.

7.2.4 Compiler support for automatic DDF registration

Currently the await clause declaring what DDFs to be read by a task is explicit. However it can be easily deduced by analyzing the code and what DDF instances are accessed in the lexical scope of a task. Then, we can delegate the populating of the await clause DDF list to the compiler which will make the jobs of implementors easier.

7.2.5 DDF data structures and Hierarchical DDFs

As we have mentioned in mapping CnC to DDFs, one can implement item collections as arrays or associative arrays of DDFs. However for particular problems a tabular structure may not be a good fit and may hamper expressibility and exacerbate possible memory footprint issues. Additionally, DDFs are expected to encapsulate fine grain data. For portable parallelism where the overhead of parallelism may be a challenge, DDFs should be a hierarchical abstraction. For example, a whole matrix of values can be one DDF, though it may be a list of column DDFs which are an array of item DDFs, where the proper granularity may be decided either at compile time or runtime. The relationship between DDFs and DDF data structures or Hierarchical DDFs may be the same between I-Vars and I-Structures.

Bibliography

[1] E. A. Lee, "The problem with threads," *Computer*, vol. 39, pp. 33–42, May 2006.

[2] OpenMP Architecture Review Board, *OpenMP Application Program Interface*, 3.0 ed., May 2008.

[3] C. Rice University, "High performance fortran language specification," *SIGPLAN Fortran Forum*, vol. 12, pp. 1–86, December 1993.

[4] G. E. Blelloch, "Nesl: A nested data-parallel language," tech. rep., Pittsburgh, PA, USA, 1992.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, (New York, NY, USA), pp. 212–223, ACM, 1998.

[6] J. Reinders, *Intel threading building blocks.* Sebastopol, CA, USA: O'Reilly & Associates, Inc., first ed., 2007.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, (New York, NY, USA), pp. 519–538, ACM, 2005.

[8] "The chapel language specification," tech. rep., February 2005.

[9] R. Barik, Z. Budimlić, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar, "The habanero multicore software research project," in *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, (New York, NY, USA), pp. 735–736, ACM, 2009.

[10] H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," *SIGART Bull.*, pp. 55–59, August 1977.

[11] R. H. Halstead, Jr., "Implementation of multilisp: Lisp on a multiprocessor," in *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, (New York, NY, USA), pp. 9–17, ACM, 1984.

[12] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer, "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, (New York, NY, USA), pp. 277–288, ACM, 2008.

[13] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşırlar, "Concurrent collections," *Sci. Program.*, vol. 18, pp. 203–217, August 2010.

[14] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, September 1999.

[15] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–12, 2009.

[16] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems," *SIGPLAN Not.*, vol. 45, pp. 341–342, January 2010.

[17] Z. Budimlić, A. Chandramowlishwaran, and K. Knobe, "Multi-core implementations of the concurrent collections programming model," *CPC'09: 14th ...*, 2009.

[18] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, (New York, NY, USA), pp. 57–76, ACM, 2007.

[19] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –12, April 2010.

[20] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," *IEEE Workload Characterization Symposium*, vol. 0, pp. 44–54, 2009.

[22] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir, "Parallel programming must be deterministic by default," in *Proceedings of the First USENIX conference*

on Hot topics in parallelism, HotPar'09, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 2009.

[23] J. Travis and J. Kring, *LabVIEW for Everyone: Graphical Programming Made Easy and Fun (3rd Edition) (National Instruments Virtual Instrumentation Series)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

[24] W. Ackerman, "Data flow languages," *Computer*, vol. 15, pp. 15 – 25, Feb. 1982.

[25] D. Gelernter, "Generative communication in linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 80–112, January 1985.

[26] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, pp. 97–107, February 1992.

[27] K. Agrawal, C. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –12, 2010.

[28] B. Liskov and L. Shrira, "Promises: linguistic support for efficient asynchronous procedure calls in distributed systems," in *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, (New York, NY, USA), pp. 260–267, ACM, 1988.

[29] P. Z. Ingerman, "Thunks: a way of compiling procedure statements with some comments on procedure declarations," *Commun. ACM*, vol. 4, pp. 55–58, January 1961.

[30] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: data structures for parallel computing," *ACM Trans. Program. Lang. Syst.*, vol. 11, pp. 598–632, October

1989.

[31] Z. Budimlić, A. M. Chandramowlishwaran, K. Knobe, G. N. Lowney, V. Sarkar, and L. Treggiari, "Declarative aspects of memory management in the concurrent collections parallel programming model," in *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, DAMP '09, (New York, NY, USA), pp. 47–58, ACM, 2008.